# Planar Graphs and Where to Find Them

Logan Stuchlik, Jordan Welch

December 11 2019

## 1 Abstract

The program takes in a user's input of number of vertices, edges, and where the edges connect to. From the given graph, the program iterates through each combination of five and six vertices and checks the adjacency matrix to that similar to the two nonplanar graphs, K5 and K3,3. The project successfully determines planarity of a graph with its connectivity checking.

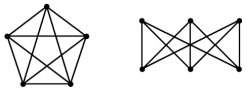
## 2 Introduction

Planar graphs have multiple applications to graph theory. Understanding the differences between planar and nonplanar graphs help for further research into graph theory. The study of planar graphs tie in the work of Euler. Euler's work is then expounded upon with the work of Kuratowski. This project explores subgraphs and subsets of the vertices of given graphs. Those subsets will be used in conjunction with the K5 and K3,3 graph.

# 3 Topic Details

Planar graphs are graphs that, given a set of vertices and edges, can be depicted with intersections happening only at vertices. These graphs can be represented on paper with all edges being straight lines. Certain planar graphs are representatives of three-dimensional shapes. In order for a graph to be planar, it cannot be (or have a subdivision of) two specific graphs. These two graphs are named K5 and K3,3 and their existence in larger graphs determine whether that larger graph is planar or not. This project works to describe the planarity of a given graph of vertices and edges.

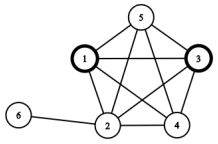
Our project begins by implementing a class graph, which contains a map of integers (used as vertices) and a vector of integers (used to show edges). The class also contains a 2D array to hold an adjacency matrix for checking connectivity. Within the class we hold Boolean functions to check against a K5 and a K3,3 graph. Those graphs are featured below:



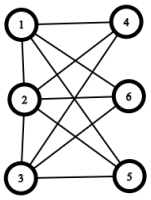
(a) $K_5$ (b) $K_{3,3}$

These two graphs are nonplanar based on the theorems explained and proved in Graphs Digraphs fourth edition (Chartrand, Lesniak, 135 - 140). The theorems explain that due to Euler's Identity, "if G is a connected plane graph with n vertices, m edges and r regions, then n – m + r = 2" (Chartrand, Lesniak, 128). K5 does not meet Euler's Identity, therefore K5 is nonplanar. In the program, the algorithm checks to see if a graph has five vertices, and from there the program directly tests the Euler Identity. A test of this scenario is shown in the output. The program runs checks for K5 and K3,3 if the graph contains more than five vertices.

The graph runs through a K5 check first and then a K3,3 check and the program returns a Boolean value for each check. The program does this based on Kuratowski's Theorem. W.T. Tutte's Graph Theory states Kuratowski's Theorem as follows, "A graph G is nonplanar if and only if it has a Kuratowski graph as a subgraph" (Tutte, 320). The program checks if there is a K5 or K3,3 subgraph to prove its planarity. The subgraph is pivotal to checking planarity. Daniel A. Marcus' Graph Theory a Problem Oriented Approach explains this by stating, "if one graph is a subdivision of another, then either both graphs are planar or else both are nonplanar" (Marcus, 81). If the program can prove any subsection or subgraph is nonplanar, then the user inputted graph is nonplanar. The program starts its K5 check by iterating through every subset of five vertices and checking their values in the adjacency matrix. If the values match those of a K5 graph, then the graph is nonplanar. An example of K5 being a subgraph appears in the program's output, and a figure shown below displays this:




The program checks for K3,3 in a similar fashion. Instead of picking every subset of five vertices, the program iterates through all subsets of six vertices within the graph. An example of a graph with a subset of K3,3 is shown in the program's output, and a figure below displays this:




The program checks for subgraphs of K5 and K3,3 and returns true when either of the subgraphs are detected.

# 4 Conclusion

The program creates a graph based on user input and outputs whether that newly created graph is planar or nonplanar. The logic within the program follows under the Euler Identity and Kuratowski's Theorem. The program directly views all possible subgraphs to prove planarity. While the algorithm is seemingly inefficient due to iterating through all possible subsets of vertices, the program shows promise of growth. Planarity is capable of expanding graph theory, and the program works to achieve determining planarity.

# 5 Bibliography

Chartrand, Gary, and Linda Lesniak. Graphs & Digraphs. 4th ed., Chapman & Hall/CRC, 2005.

Marcus, Daniel A. Graph Theory: a Problem Oriented Approach. The Mathematical Association of America, 2008.

Tutte, W.T. Graph Theory. Edited by Gian-Carlo Rota, vol. 21, Cambridge University Press, 1984.

# 6 Appendix

```
   /*
 * Logan Stuchlik & Jordan Welch
 * Project 3 - Planar Graphs and Algorithms
 * Cosc 320
 * Dec 11, 2019
 */



#include <iostream>
#include <iomanip>
#include <vector>
#include <map>
#include <queue>
#include <stdio.h>
#include <iostream>
#include <iomanip>
enum color_t {WHITE, BLACK, GRAY};
class Graph
{
  private:
    std::vector<color_t> VertColo;
    std::vector<int> VertDist;
    std::vector<int> VertPare;
    std::map<int, std::vector<int>> vertices;
```

```
    int tot_verts;
    int tot_edges;
    int **adj_matrix;
  public:
    Graph();
    Graph(int);
    void setEdge(int);
    void addVertex(int);
    bool addEdge(int, int);
    bool isPlanar();
    bool corol1_check();
    bool K5_Check();
    bool K33_Check();
    bool K33_Connect_Check(int, int, int, int, int, int);
    /*
     * We will have a typical graph structure, but here is where we deviate.
     * We will have boolean algorithms to determine if a given graph is planar or not
     * Based on different theorems and tests
     */

};


Graph::Graph() //default constructor, should never be called, but exists in case
{
  tot_verts = 0;
  tot_edges = 0;
```

```
}


Graph::Graph(int V) //non-default constructor, handles all graph creation
{
  tot_verts = V;
  adj_matrix = new int*[V+1]; //allocates memory for adj matrix
  tot_edges = 0;
  for(int i = 1; i <= V; i++)
  {
    addVertex(i);
  }

  for(int i = 0; i <= V; i++) //continues allocation
  {
    adj_matrix[i] = new int[V+1];
  }

  for(int j = 0; j <= V; j++) //initializes all of the adjacency matrix to 0, prevents crash
  {
    for(int k = 0; k <= V; k++)
    {adj_matrix[j][k] = 0;}
  }
}


void Graph::setEdge(int E) //simple edge count handler
```

```
{tot_edges = E;}



void Graph::addVertex(int addme) //adds in a vertex
{
  int space_checker = 0;
  for(auto it = vertices.begin(); it != vertices.end(); it++) //checks if vertice is already
  {
    if(it->first == addme)
    {space_checker++;}
  }

  if(space_checker != 0)
  {
    std::cout<<"Duplicate node detected. Sayonara. \n \n";
    return;
  }
  else
  {
    vertices[addme] = std::vector<int>();
  }
}



bool Graph::addEdge(int from, int to) { //adds an edge
   bool edcheck = false;
   for(auto iterat = vertices.begin(); iterat != vertices.end(); iterat++) { //checks for ed
```

```
     if(iterat->first == from) {
       for(auto iterat2 = iterat->second.begin(); iterat2 != iterat->second.end(); iterat2++
     if(*iterat2 == to) {
edcheck = true;
}
       }
     }
   }
   for(auto iterat = vertices.begin(); iterat != vertices.end(); iterat++) {
     if(iterat->first == to) {
       for(auto iterat2 = iterat->second.begin(); iterat2 != iterat->second.end(); iterat2++
     if(*iterat2 == from) {
edcheck = true;
}
       }
     }
   }
   if(edcheck) {
     std::cout<<"Preexisting" << std::endl;
     return false;
   }
   else {
   vertices[from].push_back(to);
     vertices[to].push_back(from);
     std::cout << "Edge from " << from << " to " << to << std::endl;
adj_matrix[from][to] = 1;
adj_matrix[to][from] = 1;
```

```
return true;
   }
}
```

```
bool Graph::K5_Check()
{
  for(int i1 = 1; i1 <= tot_verts - 4; i1++) { //iterates through every subset of 5 vertices
    for(int i2 = i1 + 1; i2 <= tot_verts - 3; i2++) {
      for(int i3 = i2 + 1; i3 <= tot_verts - 2; i3++) {
for(int i4 = i3 + 1; i4 <= tot_verts - 1; i4++) {
  for(int i5 = i4 + 1; i5 <= tot_verts; i5++) {
    if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i3] == 1 && adj_matrix[i1][i4] == 1 && adj_
      && adj_matrix[i2][i3] == 1 && adj_matrix[i2][i4] == 1 && adj_matrix[i2][i5] == 1 && ad
      && adj_matrix[i3][i5] == 1 && adj_matrix[i4][i5] == 1) { //checks if the connections m
      return true;
    }
  }
}
      }
    }
  }
  return false;
```

```
}


bool Graph::K33_Check()
{
  for(int i1 = 1; i1 <= tot_verts - 5; i1++) { //iterates through every subset of 6 vertices
    for(int i2 = i1 + 1; i2 <= tot_verts - 4; i2++) {
      for(int i3 = i2 + 1; i3 <= tot_verts - 3; i3++) {
for(int i4 = i3 + 1; i4 <= tot_verts - 2; i4++) {
  for(int i5 = i4 + 1; i5 <= tot_verts - 1; i5++) {
    for(int i6 = i5 + 1; i6 <= tot_verts; i6++) {
      if(K33_Connect_Check(i1, i2, i3, i4, i5, i6)) { //calls a connect check
return true;
      }
    }
  }
}
      }
    }
  }
  return false;
}


bool Graph::K33_Connect_Check(int i1, int i2, int i3, int i4, int i5, int i6) { //each if ch
  if(adj_matrix[i1][i4] == 1 && adj_matrix[i1][i5] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
    && adj_matrix[i3][i4] == 1 && adj_matrix[i3][i5] == 1 && adj_matrix[i3][i6] == 1) {
```

```
  return true;
}
if(adj_matrix[i1][i3] == 1 && adj_matrix[i1][i5] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
  && adj_matrix[i4][i3] == 1 && adj_matrix[i4][i5] == 1 && adj_matrix[i4][i6] == 1) {
  return true;
}
if(adj_matrix[i1][i3] == 1 && adj_matrix[i1][i4] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
  && adj_matrix[i5][i3] == 1 && adj_matrix[i5][i4] == 1 && adj_matrix[i5][i6] == 1) {
  return true;
}
if(adj_matrix[i1][i3] == 1 && adj_matrix[i1][i4] == 1 && adj_matrix[i1][i5] == 1 && adj_ma
  && adj_matrix[i6][i3] == 1 && adj_matrix[i6][i4] == 1 && adj_matrix[i6][i5] == 1) {
  return true;
}
if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i5] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
  && adj_matrix[i4][i2] == 1 && adj_matrix[i4][i5] == 1 && adj_matrix[i4][i6] == 1) {
  return true;
}
if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i4] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
  && adj_matrix[i5][i2] == 1 && adj_matrix[i5][i4] == 1 && adj_matrix[i5][i6] == 1) {
  return true;
}
if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i4] == 1 && adj_matrix[i1][i5] == 1 && adj_ma
  && adj_matrix[i6][i2] == 1 && adj_matrix[i6][i4] == 1 && adj_matrix[i6][i5] == 1) {
  return true;
}
if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i3] == 1 && adj_matrix[i1][i6] == 1 && adj_ma
```

```
    && adj_matrix[i5][i2] == 1 && adj_matrix[i5][i3] == 1 && adj_matrix[i5][i6] == 1) {
    return true;
  }
  if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i3] == 1 && adj_matrix[i1][i5] == 1 && adj_ma
    && adj_matrix[i6][i2] == 1 && adj_matrix[i6][i3] == 1 && adj_matrix[i6][i5] == 1) {
    return true;
  }
  if(adj_matrix[i1][i2] == 1 && adj_matrix[i1][i3] == 1 && adj_matrix[i1][i4] == 1 && adj_ma
    && adj_matrix[i6][i2] == 1 && adj_matrix[i6][i3] == 1 && adj_matrix[i6][i4] == 1) {
    return true;
  }
  return false;
}


bool Graph::corol1_check() //This uses corollary 1
{
  int Math = 3 * tot_verts - 6;
  if(tot_edges <= Math)
  {return true;}
  else
  {return false;}
}


bool Graph::isPlanar()
{
```

```
  if(tot_verts < 5)
  {return true;}
  //A graph with less than 5 edges is always planar since it can never be K5 or K3,3

  if(tot_verts == 5)
  {
    if(!corol1_check())
    {return false;}
    else
    {return true;}
  }

  if(K5_Check())
  {return false;}

  if(K33_Check())
  {return false;}

  return true;
}

int main()
```

```
{
  int Jo_Lo = 21;
  while(Jo_Lo < 1 || Jo_Lo > 20) {
    std::cout << "\nPlease input how many vertices you want (max 20): ";
    std::cin >> Jo_Lo;
  }
  Graph Guraph(Jo_Lo);
  std::cout << "\nThank you.";
  int Dan_Gan = ((Jo_Lo * (Jo_Lo - 1)) / 2) + 1;
  int R_G = Dan_Gan - 1;
  //n(n-1)/2
  while(Dan_Gan < 0 || Dan_Gan > R_G) {
    std::cout << "\nPlease input how many edges you want (max "<< R_G << "): ";
    std::cin >> Dan_Gan;
  }
  Guraph.setEdge(Dan_Gan);
  int ver1, ver2;
  for(int i = 1; i <= Dan_Gan; i++) {
    ver1 = ver2 = 0;
    while((ver1 < 1 || ver1 > Jo_Lo) && (ver2 < 1 || ver2 > Jo_Lo))
    {
      std::cout << "\nPlease enter in two vertices to connect an edge to (1 - " << Jo_Lo <<
      std::cin >> ver1 >> ver2;
    }
    if(!Guraph.addEdge(ver1, ver2)) {
      i--;}
  }
```

```
  if(Guraph.isPlanar())
  {std::cout<<"\nHooray! Your graph is planar!! :)\n";}
  else
  {std::cout<<"\nAwwwww! Your graph is not planar.. :( big sad\n";}


  return 0;
}
```