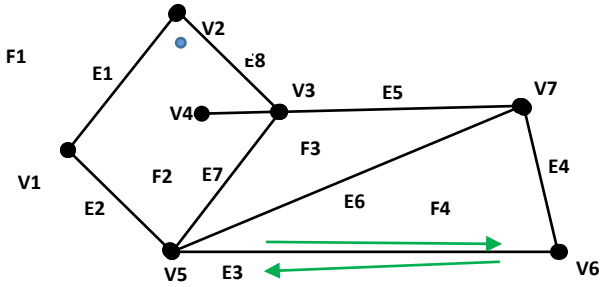
Title: Planar Graph

Team Members: Donovan hatch, Deep Ghosh

Graphs: A graph is a representation of objects as nodes where some of them are connected by links.

Planar Graphs: A graph which lies on the same plane and no two edges intersect each other is called a planar graph. All edges are straight lines in a planar graph.

Example:



Terminology: For the rest of this report we will be using **F** to denote faces of a graph, **V** to denote vertices of a graph and **E** to denote edges of a graph.

Euler Invariant: Euler Invariant is a formula which describes the shape or structure of a topology regardless of the way it is bent. The formula is denoted by,

$$F + V = E + b \text{ where } b = 2 \text{ for convex surfaces.}$$

In the following part of the document we will be looking to define different data structures to represent a planar polygonal graph.

Assumptions: We will always follow the edges inside a closed face of a graph (e.g. F3, F4, F2) clockwise. Edges when followed on the outer face (e.g. F1) will be followed counter clockwise.

1. **Pointer based methodology**:
   This methodology is the old method where graph elements were used to be stored using pointers. Let us take our example graph to elaborate on this method.
   All the vertex information regarding position, edges radiating out of a vertex, the next edges (clockwise edge) and previous edges (counter clockwise edge) are stored. Pointers are used to point to find the next item in the list.
   This methodology has proved to be very memory intensive and unwieldy. Implementation is easier as it is more developer friendly.
2. **Index based methodology:**
   This methodology has several subsections. The general idea under this methodology is to use arrays to store the data and use the indexes of the arrays to link connected elements of the graph. The main aim of this methodology is to reduce the memory consumption of the implemented data structure.
   The first form of implementation (winged edge data structure) uses arrays to

store the edges start and end vertex in two different arrays. The two array elements are linked by their indices. E.g In the sample graph for edge E2 arrStart[2] = V1 and arrEnd[2] = V5.

This method is good for direct access to a particular edge and random access to any edge. This is because the indices allows to access any element faster. Also it allows iteration over vertices or edges as required. On the downside traversal from one edge to other is expensive. Also finding or traversing faces is very difficult and ambiguous, as no face information is stored.

Now we are going to introduce a new data structure which uses Constant Average/Amortized Time.

We will use a new concept of Half Edges to address the problems of the previous data structure. Half Edges or **H** comes from the fact that each edge in a graph can have only 2 faces adjacent to it. For each of the faces the edge will follow a different direction (opposite to each other). So instead of denoting edges in data structure we will define two half edges (one for each face) for each edge. By doing so we can follow the Half Edges around to define each face (the green arrows in the sample diagram).

Based on the above mentioned definitions we can define a graph using the set of following arrays.

G (for vertex position), **V** (for storing starting points of each half edge) and **NC** (for storing the next start position of a half edge for the corresponding value in **V**). We will define another term as **corners.** Corner is a point near a vertex that represents line connections for a face with a direction. In other words the index of array **V** is the corner id (the blue dot represents a single corner in the sample diagram). Using the above data structure we can address the problems of the previous data structure. For a hanging edge there is only one corner for the hanging vertex. All other vertices have min of 2 corners. We can navigate the faces by following the half edges. Based on the half edge data structure we can define the following functions on corners to move around the graph.

1. Next → Moves to the next corner on the face of the current corner.
2. Prev → Moves to the previous corner on the face of the current corner.
3. Swing → Moves to the next corner in the adjacent face of the current corner.
4. Unswing → Moves to the corner which swings to this corner.
4. Z → Moves to the swing corner of the next corner of the current corner.

Tip: Corners can be identified when adding an edge by finding the angle between the existing edges and the edge being added to a randomly selected existing edge.