Graphs and Shortest Paths

October 24, 2011

Notes

- Practice Problems:

23.1	2, 3, 5, **7**, 8-16, 20
23.2	1-7, 12, 14, 15, 16
23.3	1-7

Dijkstra's Algorithm for Shortest Paths

Edsger Wybe Dijkstra (1930–2002), 1318 unpublished manuscripts, University of Texas professor.

Labeling procedure for a vertex v:

(PL) permanent label: length L_v of a shortest path $1 \to v$ is found.

(TL) temporary label: upper bound L'_v for the length of a shortest path $1 \to v$ is found.

We are going to have two sets of vertices at each step:

- $\mathcal{P}$ the vertices with a permanent label.
- $\mathcal{T}$ the vertices with a temporary label.

Dijkstra's Algorithm for Shortest Paths

Algorithm Dijkstra's [G = (V, E)]

INPUT: Connected graph G = (V, E) with a origin 1.
OUTPUT: Lengths L_i of the shortest paths from $1 \to i$.

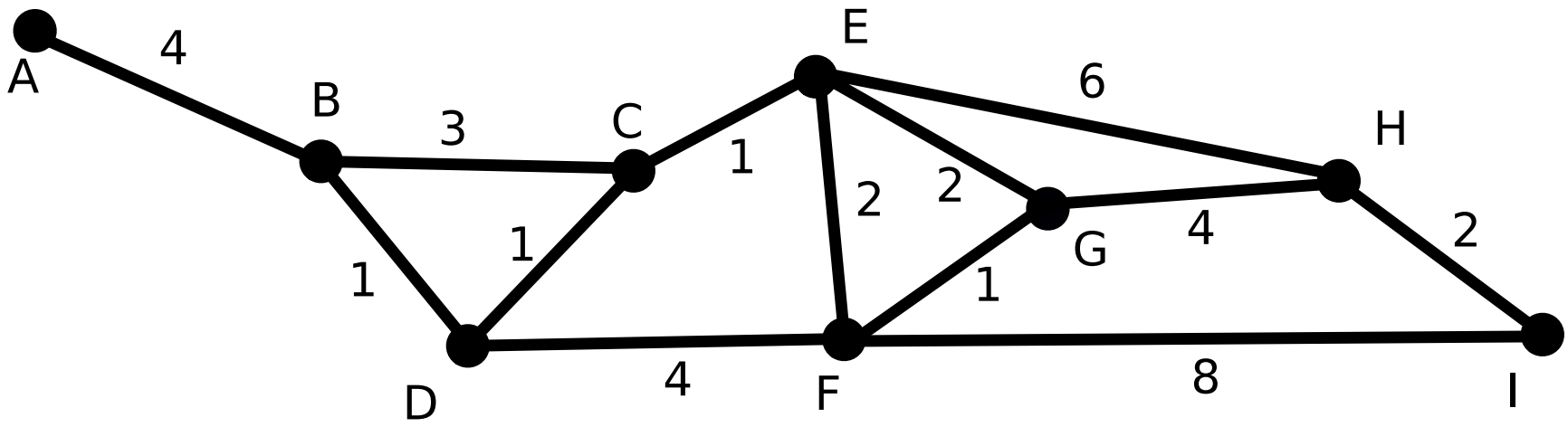
1. Initial Step: Vertex 1 is added to $\mathcal{P}$ with $L_1 = 0$. Vertex j is added to $\mathcal{T}$ with label $L'_j = \ell_{1j}$.
2. Find a k in $\mathcal{T}$ for which L'_k is the smallest (take the smallest k if there are several).

   Set $L_k = L'_k$ and move k from $\mathcal{T}$ to $\mathcal{P}$.

   IF $\mathcal{T}$ is empty then OUTPUT $L_1, L_2, \ldots, L_n$ STOP

   ELSE CONTINUE
3. *Updating the Temporary Labels*: For all $j \in \mathcal{T}$, set $L'_j = \min(L'_j, L_k + \ell_{kj})$. GO TO Step 2.
   END Dijkstra

Dijkstra's Algorithm Example



23.4 Shortest Spanning Trees

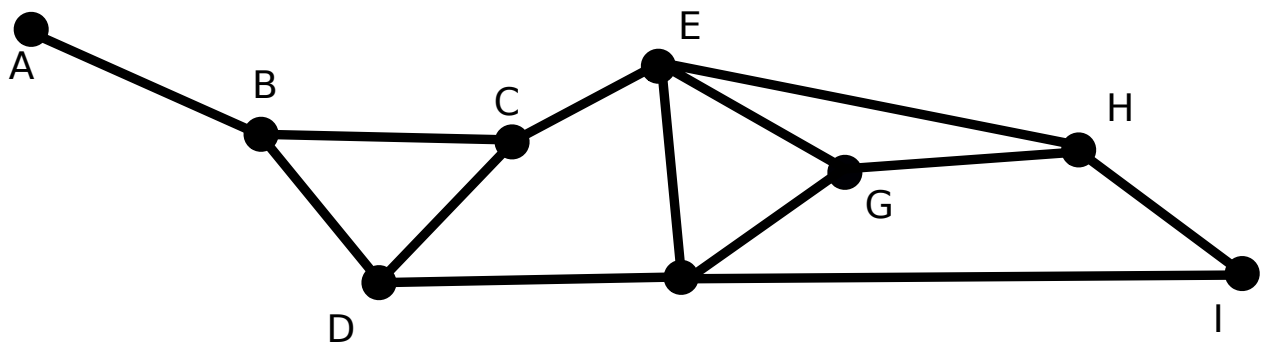
In many applications it is helpful to find the smallest network necessary for connecting each vertex. For example in building a subway line to connect points of interest in a city.

A **tree** is a graph that is connected and has **no** cycles.

A **Connected** graph is a graph which has a path connecting any two vertices.

A **Spanning Tree** in a given connected graph is a tree containing all the vertices of the graph.

A spanning tree with n vertices has $n-1$ edges. There can be several spanning trees in a graph.



23.4 Shortest Spanning Trees

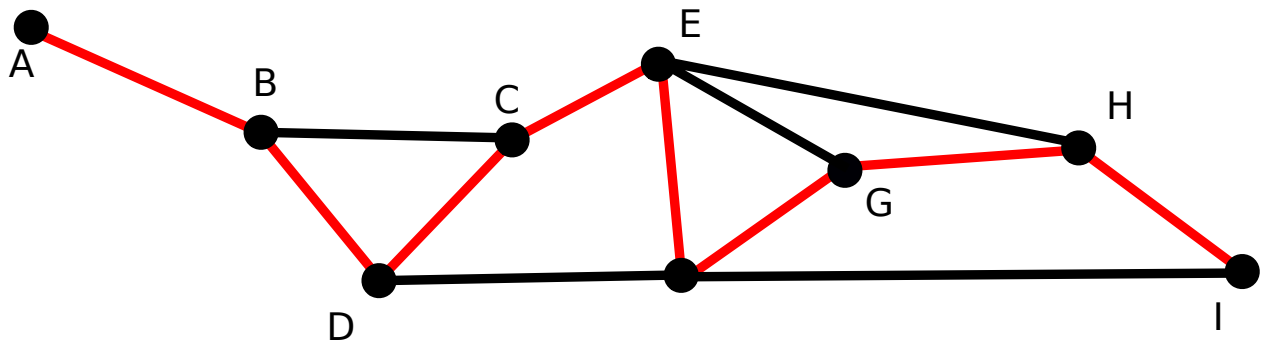
In many applications it is helpful to find the smallest network necessary for connecting each vertex. For example in building a subway line to connect points of interest in a city.

A **tree** is a graph that is connected and has **no** cycles.

A **Connected** graph is a graph which has a path connecting any two vertices.

A **Spanning Tree** in a given connected graph is a tree containing all the vertices of the graph.

A spanning tree with n vertices has $n-1$ edges. There can be several spanning trees in a graph.



23.4 Shortest Spanning Trees

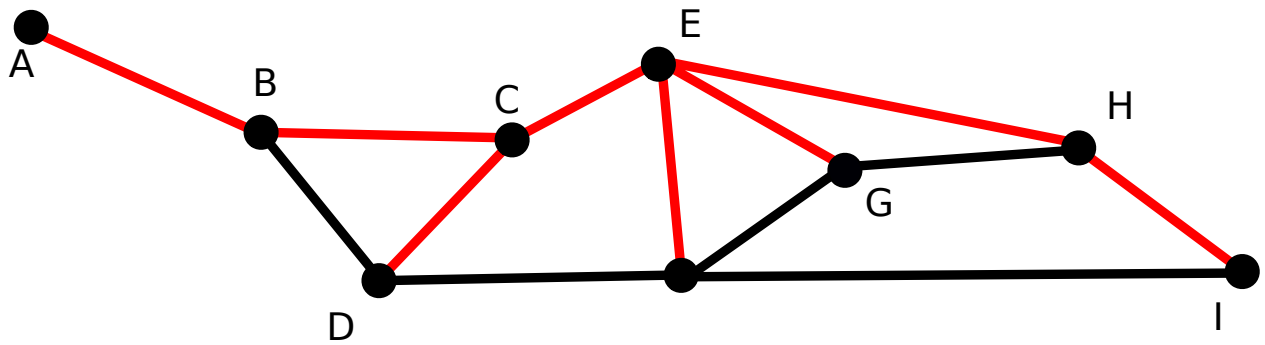
In many applications it is helpful to find the smallest network necessary for connecting each vertex. For example in building a subway line to connect points of interest in a city.

A **tree** is a graph that is connected and has **no** cycles.

A **Connected** graph is a graph which has a path connecting any two vertices.

A **Spanning Tree** in a given connected graph is a tree containing all the vertices of the graph.

A spanning tree with n vertices has $n-1$ edges. There can be several spanning trees in a graph.



Shortest Spanning Tree

A **shortest spanning tree** T in a connected graph G (whose edges of lengths ℓ_{ij}) is a spanning tree for which $\sum \ell_{ij}$ (total length of edges) is minimum compared to other spanning trees.

Simple Fact: The set of shortest paths from vertex 1 to all the other vertices forms a spanning tree.

The key facts are that the set of all shortest paths will obviously reach all vertices, and by Bellman's principle it would not make sense for it to include cycles.

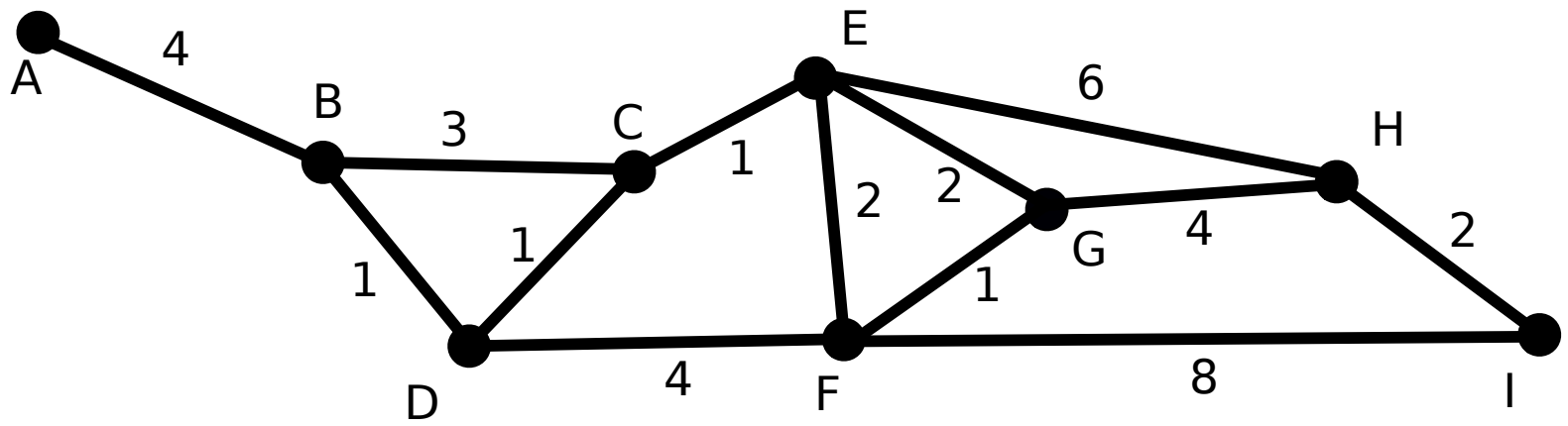
Kruskals Greedy Algorithm for Finding Shortest Spanning Trees

Algorithm Kruskal [G = (V, E)]

INPUT: Connected graph G = (V, E).
OUTPUT: A Shortest Spanning Tree T in graph G.

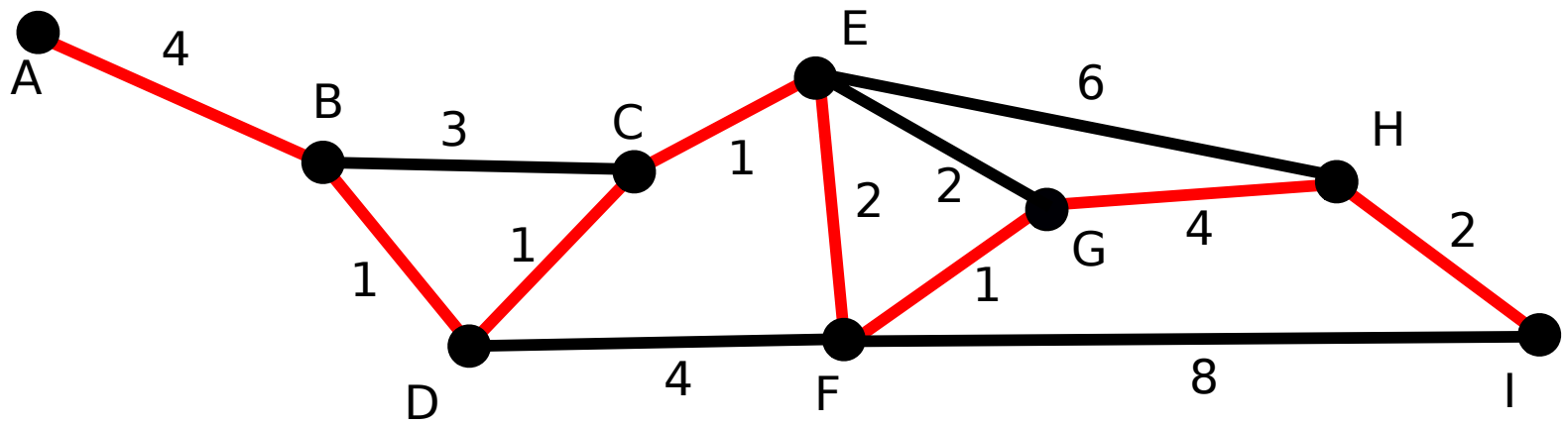
1. Order the edges of G in ascending order of length.
2. Choose them in this order as edges of T, rejecting an edge only if it forms a cycle with the edges already choosen.
3. If $n-1$ edges have been choosen, then OUTPUT T (the set of edges) STOP.
   END Kruskal

Example: Kruskal



Edges in Order: BD, CD, CE, FG, EF, EG, HI, BC, AB, DF, GH, EH, FI.

Example: Kruskal



Edges in Order: BD, CD, CE, FG, EF, EG, HI, BC, AB, DF, GH, EH, FI.

23.5 Prims Algorithm for Shortest Spanning Trees

The problem with Kruskal's Greedy algorithm is that it is a top down approach. Note that in the example that we did there were a number of edges we eventually rejected before we found the final tree. Also notice that part way through the algorithm we had a disconnected tree.

Prims method is to grow the spanning tree so that if we stop the algorithm early, while we will not have spanning, we will always have a tree.

The is particularly helpful for an exceptionally large graph, and in the end the type of graphs we care about in the wild are always exceptionally large.

The algorithm was discovered in 1930 by mathematician Vojtech Jamik, and later independently by computer scientists Robert C. Prim (1957), and also independently by Dijkstra in 1959. Therefore it is sometimes called the DJP or Janik Algorithm.

Prims Algorithm for Shortest Spanning Trees

Algorithm Prim [G = (V, E)]

INPUT: Connected graph G = (V, E).
OUTPUT: A Shortest Spanning Tree T in graph G.

1. Initial step: Set $i(k) = 1$, $U = \{1\}$, and $S = \emptyset$.

   Label vertex k with $\lambda_k = \ell_{ik}$.

2. Addition of an edge to the tree T: Let λ_j be the smallest λ_k for k not in U. Include vertex j in U and edge $(i(j), j)$ in S.

   If $U = V$ then compute $L(T) = \sum \ell_{ij}$ (summing over S)

   OUTPUT S, L(T). STOP.

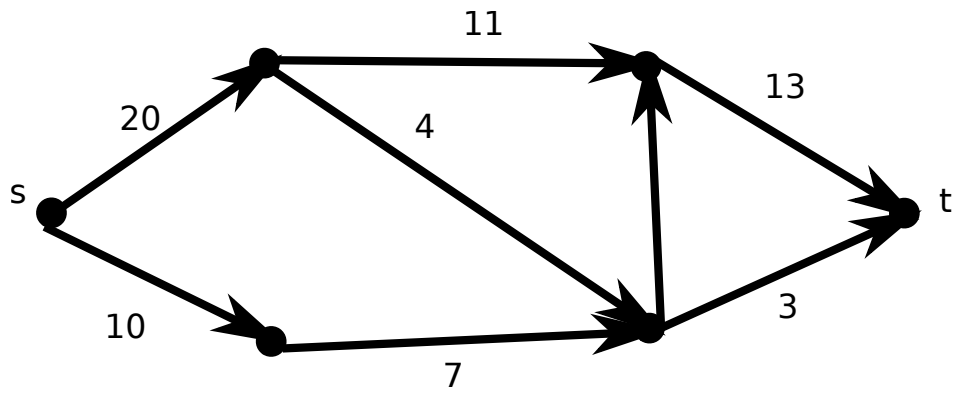
   ELSE continue.

3. Label Updating: For every k not in U if $\ell_{jk} < \lambda_k$ then set $\lambda_k = \ell_{jk}$ and $i(k) = j$. GO TO Step 2.
   END Prim

23.6 Flows in Networks

Another type of combinatoric optimization is to maximize the flow through a network. Here we turn from graphs to digraphs.

The weights on the edges of the network now refer to the capacity of the network (think in terms of **electric current**, **water**, **traffic**, **information**, etc.



Maximum Flow

The problem is to determine the maximum flow allowed in this network from $s \rightarrow t$.

Flow Augmenting Paths

Given a network with a capacity and flow, we will attempt to look for paths in the network from $s \to t$ which can be used to increase the net-flow.

Let f_{ij} be the flow rate from i to j and c_{ij} be the capacity.

Flow Augmenting Path

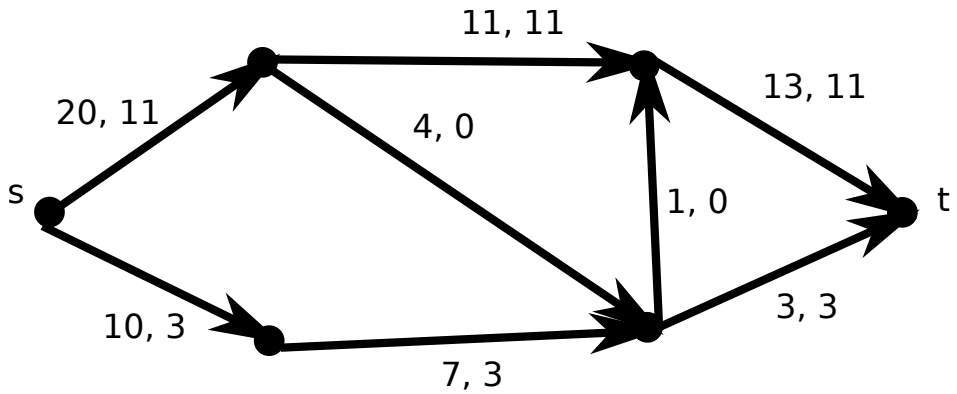
Is a path from $s \to t$ such that

- No forward edge is used to capacity $f_{ij} < c_{ij}$
- No backward edge has zero flow $f_{ij} > 0$.

The point is that a **Flow Augmenting Path** represents unused capacity.

Example: Flow Augmenting Path

Find flow augmenting paths in the network:



where the first number is the capacity, and the second is the flow.

Cut Set

A **Cut Set** of a network is a collection of edges which divide the graph into two components with s in one and t in the other. The idea should be very natural: If we want to know what is flowing from s to t, we should cut the network and look in the 'pipes'.

Theorem

Net Flow in Cut Sets Any given flow in the network G is the net flow through any cut set of G.

Theorem

Upper Bound for Flows A flow in the network G cannot exceed the capcity of any cut set in G.

The main theorem is then

Theorem

A flow from $s \to t$ in a netowrk G is maximum iff there does not exist a flow augmenting path $s \to t$ in G.

Proof.

Suppose there exists a flow augmenting path. The flow augmenting path represents unused capacity and we can increase the flow along this path until it is no longer flow augmenting. (adding to f_{ij} when the arrow goes with the path, decreasing f_{ij} when the arrow goes against). Thus the flow could not have been maximum.

The main theorem is then

Theorem

A flow from $s \to t$ in a netowrk G is maximum iff there does not exist a flow augmenting path $s \to t$ in G.

Proof.

For the other direction suppose there is not a flow augmenting path. Let S_0 be the set of all vertices in G such that there is a flow augmenting path to i from s, and let T_0 be the set of all other vertices. Consider any edge (i, j) in G with $i \in S_0$ and $j \in T_0$. We have a flow augmenting path from $s \to i$ but not from $s \to i \to j$. Thus we must have

$$f_{ij} = \begin{cases} c_{ij} & \text{if } (i,j) \text{ is forward} \\ 0 & \text{if } (i,j) \text{ is backward} \end{cases}$$

The main theorem is then

Theorem

A flow from $s \to t$ in a netowrk G is maximum iff there does not exist a flow augmenting path $s \to t$ in G.

Proof.

The sets S_0 and T_0 give a cut set for the graph, and thus by the previous Theorems the net flow f is given by the sum of the f_{ij} between S_0 and T_0 and any other flow is less than or equal to f. □

Ford Fulkerson Algorithm

The Ford Fulkerson Algorithm for finding the maximum flow of a network, which we will not go into formally, relies on the principle of flow augmenting paths. The algorithm searches our spare capacity in the network by looking for flow augmenting paths and then increasing the flow along those.

23.8 Graph Coloring

One of the initial problems we began with was a scheduling problem.
You are working for a company which is building an electronic device. When the device is activated it spends one minute making a number of measurements and computations which have to occur at specific times during that minute:

Task	A	B	C	D	E
Time Interval	(0, 20)	(10, 20)	(10, 30)	(20, 60)	(20, 40)
	F	G	H		
	(30, 45)	(40, 50)	(45, 60)		

What is the minimum number of processors we will need to include in this device to handle these 8 tasks?

Graph Coloring

Label each task as a vertex and connect vertices whose time intervals overlap.

Overlapping vertices cannot use the same processors, so the question becomes: How many colors do we need so that each vertex of the graph has a color, and no two adjacent vertices have the same color?

Famous Example

The Four-Color Problem

Any map only requires four colors to differentiate the countries.
No two adjacent countries will have the same color.

This problem was proposed in 1852, yet despite numerous incorrect proofs a correct proof was not given until 1976 by Appel and Hanken.

The reduced the problem down to a set of 1936 maps would could have a counter example, by showing that any map had a portion that looked like one of these.

They then used a computer to exhaustively demonstrate that four colors were sufficient for each of those 1936 maps.

This was the first major theorem whose proof relied on the use of computers!!
In 2005 a simpler proof was found by Gonthier using his Theorem Proving software (no tricks, just logic).

My work

Lots of interesting questions now occur: Suppose instead of a sphere, the earth was shaped as a doughnut, how many colors would maps require then?
Suppose the topology was more complicated: A multi-holed doughnut? A surface with a singularity?

Random Graphs, Random Matrices

While we are here, let us mention the idea of studying random graphs. Typical graphs of which we are concerned are quite large. Often it is impossible to know for example, all of the lengths of the graph precisely, or even to study all of the connections precisely.

The field of random graphs has been developed to study large graphs with imperfect resolution.

Going back to the adjacency matrix, this leads naturally to the question of how random matrices behave, particularly as the size of the matrix becomes large.

Random Graphs, Random Matrices

Finally this leaves us with an interesting question about graphs: What do the eigenvalues of the adjacency matrix tell us about the graph?

We don't know much. There is *some* connection to the cycles of the graph.

Interested? Come see me and we can talk about a project for you to work on.