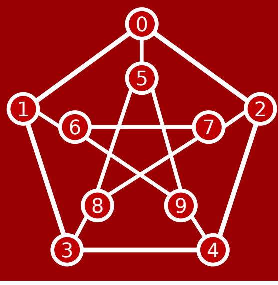
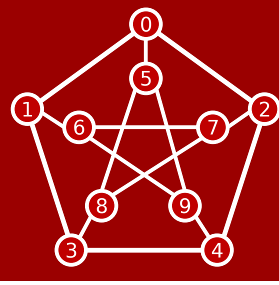
15-251: Great Theoretical Ideas in Computer Science
Lecture 10

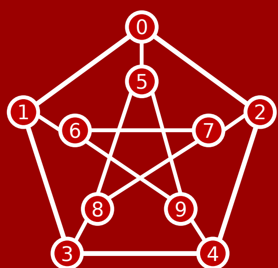
Graphs: The Basics



What is a graph?



What isn't a graph?!



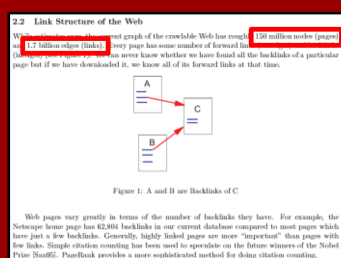
Facebook

Vertices = people   Edges = friendships

Facebook

vertices $n \approx 10^9$ # edges $m \approx 10^{12}$

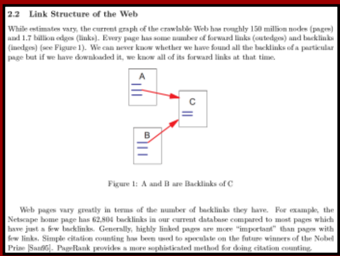
World Wide Web



1998 paper
on PageRank

Vertices = pages   Edges = hyperlinks
("directed graph")

World Wide Web


2.2 Link Structure of the Web

While estimates vary, the current graph of the crawlable Web has roughly 150 million nodes (pages) and 1.7 billion edges (links). Every page has some number of forward links (outedges) and backlinks (inedges) (see Figure 1). We can never know whether we have found all the backlinks of a particular page but if we have downloaded it, we know all of its forward links at that time.

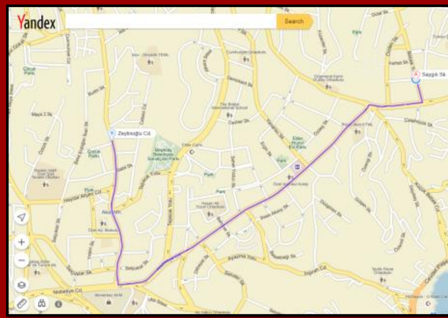
Figure 1: A and B are Backlinks of C

Web pages vary greatly in terms of the number of backlinks they have. For example, the Netscape home page has 62,804 backlinks in our current database compared to most pages which have just a few backlinks. Generally, highly linked pages are more "important" than pages with few links. Simple citation counting has been used to speculate on the future winners of the Nobel Prize [San95]. PageRank provides a more sophisticated method for doing citation counting.

1998 paper on PageRank

Today: Perhaps $n \approx 10^{12}$, $m \approx 10^{13}$?

Street Maps



Vertices = intersections Edges = streets

Zachary Karate Club

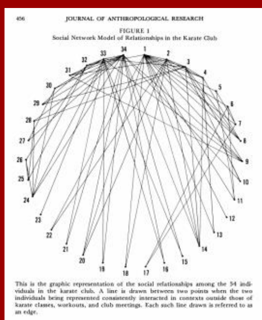

456 JOURNAL OF ANTHROPOLOGICAL RESEARCH

FIGURE 1
Social Network Model of Relationships in the Karate Club

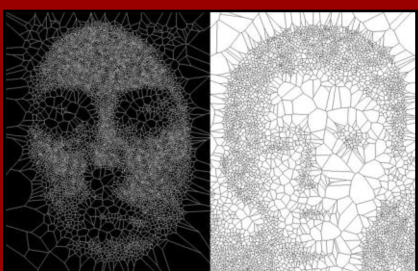
This is the graphic representation of the social relationships among the 34 individuals in the karate club. A line is drawn between two points when the two individuals being represented consistently interacted in contexts outside those of karate classes, workouts, and club meetings. Each such line drawn is referred to as an edge.

34 vertices (karatekas) 78 edges (friendships)

Zachary Karate Club CLUB

(networkkarate.tumblr.com)

Graphs from images



These are "planar" graphs; drawable with no crossing edges.

Register allocation problem

A compiler encounters:

```
temp1 := a+b
temp2 := -temp1
    c := temp2+d
```

5 variables; can it be done with 4 registers?

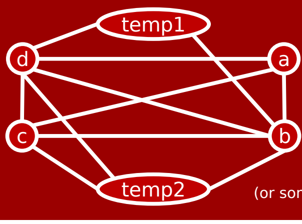
G. Chaitin (IBM, 1980) breakthrough:
Let variables be vertices. Put edge between u and v if they need to be live at same time. The least number of registers needed is the chromatic number of the graph.

Register allocation problem

A compiler encounters:

```
temp1 := a+b
temp2 := −temp1
    c := temp2+d
```

5 variables; can it be done with 4 registers?



(or something like that)

Computer Science Life Lesson:

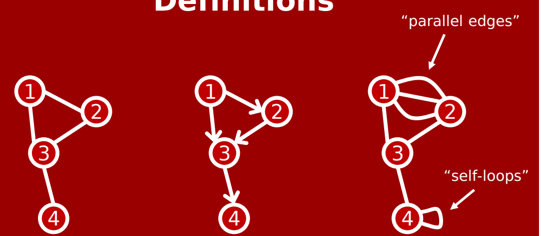
If your problem has a graph, ☺.
If your problem doesn't have a graph,
try to make it have a graph.

Warning:

The remainder of the lecture is,
like, 100 definitions.

If you've seen them all before 10 times,
play **`http://planarity.net`** on your phone.

Definitions



Simple Undirected Graphs

Directed Graphs

General Graphs

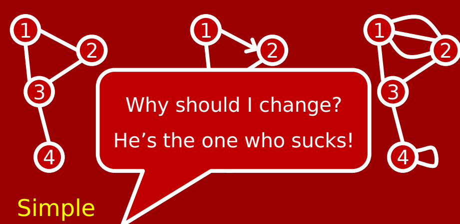
(AKA annoying graphs)

Definitions

Acoustic Guitar

Electric Guitar

Definitions



Simple Undirected Graphs

Directed Graphs

General Graphs

(AKA annoying graphs)

Definitions

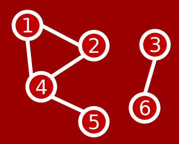
A **graph** G is a pair (V,E) where:
V is the finite set of **vertices/nodes**;
E is the set of **edges**.

Each edge $e \in E$ is a pair $\{u,v\}$,
where $u,v \in V$ are distinct.

Example:
V = {1,2,3,4,5,6}
E = { {1,2}, {1,4}, {2,4}, {3,6}, {4,5} }

Definitions

G = (V,E) can be drawn like this:



Example:
V = {1,2,3,4,5,6}
E = { {1,2}, {1,4}, {2,4}, {3,6}, {4,5} }

Notation

n almost always denotes |V|

m almost always denotes |E|

Edge cases (haha)

Question:
Can we have a graph with no edges (m=0)?

Answer:
Yes! For example,
V = {1,2,3,4,5,6}
E = ∅

Called the "**empty graph**" with n vertices.

Edge cases

Question:
Can we have a graph with no **vertices**?

Answer:
Um...... well......

IS THE NULL-GRAPH A POINTLESS CONCEPT?

Frank Harary
University of Michigan
and Oxford University

Ronald C. Read
University of Waterloo

ABSTRACT

The graph with no points and no lines is discussed critically. Arguments for and against its official admittance as a graph are presented. This is accompanied by an extensive survey of the literature. Paradoxical properties of the null-graph are noted. No conclusion is reached.

Edge cases

Question:
Can we have a graph with no **vertices**?

Answer:
It's to convenient to say **no**.
We'll require $V \neq \varnothing$.

One vertex ($n = 1$) definitely allowed though.
Called the "**trivial graph**".

1

More terminology

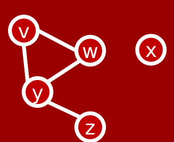
Suppose $e = \{u,v\} \in E$ is an edge.

We say:
u and v are the **endpoints** of e,
u and v are **adjacent**,
u and v are **incident** on e,
u is a **neighbor** of v,
v is a **neighbor** of u.

More terminology

For $u \in V$ we define $N(u) = \{v : \{u,v\} \in E\}$,
the **neighborhood** of u.

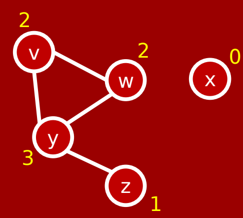
E.g., in the below graph, $N(y) = \{v,w,z\}$,
$N(z) = \{y\}$,
$N(x) = \varnothing$.



The **degree** of u is
$\deg(u) = |N(u)|$.

E.g., $\deg(y)=3$, $\deg(z) = 1$, $\deg(x) = 0$.

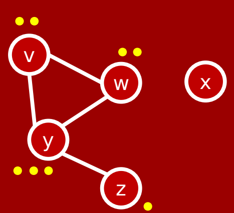
Theorem:
Let $G = (V,E)$ be a graph. Then $\sum_{u \in V} \deg(u) = 2|E|$.



$2+2+0+3+1 = 8$
$= 2 \cdot 4$

✓

Theorem:
Let $G = (V,E)$ be a graph. Then $\sum_{u \in V} \deg(u) = 2|E|$.



$2+2+0+3+1 = 8$
$= 2 \cdot 4$

✓

Remark: Classic "double counting" proof.

Proof of $\sum_{u \in V} \deg(u) = 2|E|$ **:**

Tell each vertex to put a "token" on each edge it's incident to.
Vertex u places $\deg(u)$ tokens. So one hand,

total number of tokens = $\sum_{u \in V} \deg(u)$.

On the other hand, each edge ends up with exactly 2 tokens, so

total number of tokens = $2|E|$.

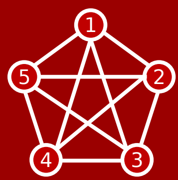
Therefore $\sum_{u \in V} \deg(u) = 2|E|$. $\blacksquare$

Question:

In an n-vertex graph, how large can m be?
(That is, what is the max number of edges?)

Answer: $\binom{n}{2} = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2)$

E.g.: $n = 5$, $m = \binom{5}{2} = 10$.



Called the **complete graph** on n vertices. Notation: K_n

A bogus “definition”

If $m = O(n)$ we say G is “**sparse**”.
If $m = \Omega(n^2)$ we say G is “**dense**”.

This does not actually make sense.
E.g., if $n = 100$, $m = 1000$, is it sparse or dense? Or neither?

It **would** make sense if you had a **sequence** or **family** of graphs.

Anyway, it’s handy **informal** terminology.

Let’s go back to talking about K_n.

In K_n, every vertex has the **same degree**.

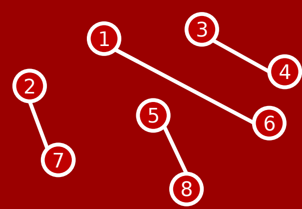
This is called being a **regular** graph.

We say G is **d-regular** if all nodes have degree d.

For example: K_n is $(n-1)$-regular;
the empty graph is 0-regular.

What about d-regular for other d?

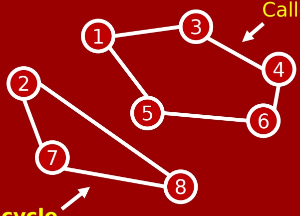
1-regular graphs



Possible if and only if $|V|$ is **even**.

Such a graph is called a **perfect matching**.

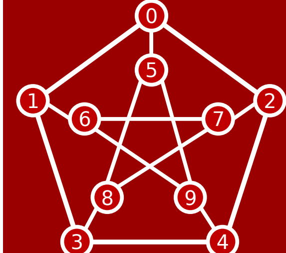
2-regular graphs

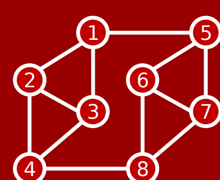


2-regular graph is a disjoint collection of cycles.

3-regular graphs

There are lots and lots of possibilities.





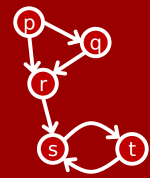
A little about "directed graphs"

First, they have a "celebrity couple"-style nickname, a la:


"Kimye"


"Brangelina"

A little about "directed graphs"



"Digraph"

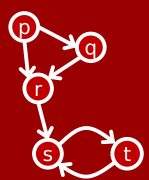
Now an edge is an **ordered** pair, $e = (u,v)$.

$G = (V,E)$, where:

$V = \{p,q,r,s,t\}$

$E = \{ (p,q), (p,r), (q,r), (r,s), \underbrace{(s,t), (t,s)}_{\text{these are distinct edges}} \}$

A little about "directed graphs"



Now there's **out-degree** and **in-degree**

$\deg_{out}(u) = |\{v : (u,v) \in E\}|$

$\deg_{in}(u) = |\{v : (v,u) \in E\}|$

E.g.: $\deg_{out}(p) = 2$ $\deg_{out}(s) = 1$
$\deg_{in}(p) = 0$ $\deg_{in}(s) = 2$

Storing graphs on a computer

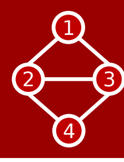
Two traditional methods:

Adjacency Matrix

Adjacency List

For both, assume $V = \{1, 2, \ldots, n\}$.

Our example graph:



Adjacency Matrix

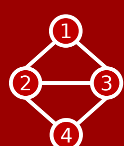
Adjacency matrix A is $n \times n$ array.

$$A[i,j] = \begin{cases} 1 & \text{if } i, j \text{ are adjacent} \\ 0 & \text{if } i, j \text{ not adjacent} \end{cases}$$

For digraphs, put 1 iff $i \to j$ is an edge.

For general graphs, put # edges $i \to j$.

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



Adjacency Matrix

Pros:

Extremely simple.

$O(1)$ time lookup for whether edge is present/absent.

Can apply linear algebra to graph theory… hmm…

Cons:

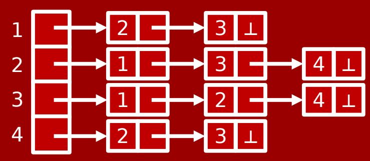
Always uses n^2 space (memory).
Very wasteful for "sparse" graphs ($m \ll n^2$).

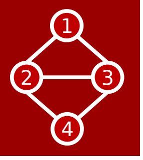
Takes $\Omega(n)$ time to enumerate neighbors of a vertex.

Adjacency List

A length-n array Adj, where Adj[i] stores a pointer to a **list** of i's neighbors.

Adj =





Adjacency List

Pros:

Space-efficient. Memory usage is… O(n) + O(m)

Efficient to run through neighbors of vertex u:
O(deg(u)) time.

Cons:

Single edge lookup can be slow:
To check if (u,v) is an edge, may take Ω(deg(u)) time, which could be Ω(n) time.

Storing graphs on a computer

Any other possibilities? Sure!

Adjacency matrix and list were good enough for your grandparents.

But you could do something new and fresh. Maybe add in a hash table to your adj. list.

Time for **more definitions**! Yay!

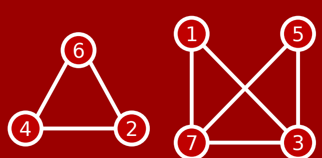
Let's talk about **connectedness**.

Here's a graph G = (V,E):

V = {1,2,3,4,5,6,7}

E = { {1,3}, {1,7}, {2,4}, {2,6}, {3,5}, {3,7}, {4,6}, {5,7} }

Notice anything peculiar about it?



This graph is **not connected**.

Terminology

A graph G = (V,E) is **connected** if
∀ u,v ∊ V, v is **reachable** from u.

Vertex v is **reachable** from u if
there is a **path** from u to v.

That's correct, but let's say instead:
"if there is a **walk** from u to v".

Terminology

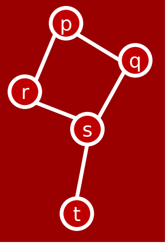
A **walk** in G is a sequence of vertices

$v_0, v_1, v_2, \ldots, v_n$ (with $n \geq 0$)

such that $\{v_{t-1}, v_t\} \in E$ for all $1 \leq t \leq n$.

We say it is a walk **from** v_0 **to** v_n, and its **length** is n.

Example:
(p, q, s, r, p, r, s, t) is a **walk from p to t of length 7**.



Terminology

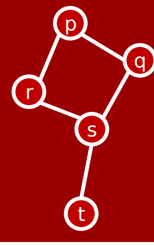
A **walk** in G is a sequence of vertices

$v_0, v_1, v_2, \ldots, v_n$ (with $n \geq 0$)

such that $\{v_{t-1}, v_t\} \in E$ for all $1 \leq t \leq n$.

Question:
Is vertex u reachable from u?

Answer:
Yes.
Walks of length 0 are allowed.



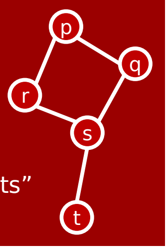
Terminology

A **path** in G is a walk with no repeated vertices.

Fact:
There is a walk from u to v
iff there is a path from u to v.

Because you can always "shortcut" any repeated vertices in a walk.

Example:
walk (p, q, s, r, p, r, s, t) "shortcuts" to path (p, q, s, t).

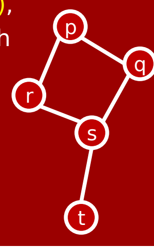


Terminology

A **path** in G is a walk with no repeated vertices.

If v is reachable from u, we define the **distance from u to v**, $\mathrm{dist}(u,v)$, to be the length of the shortest path from u to v.

Examples:
$\mathrm{dist}(p,r) = 1$, $\mathrm{dist}(p,s) = 2$,
$\mathrm{dist}(p,t) = 3$, $\mathrm{dist}(p,p) = 0$.

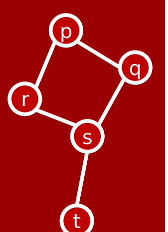


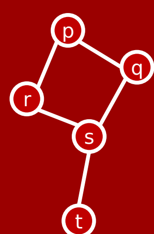
Terminology

A **path** in G is a walk with no repeated vertices.

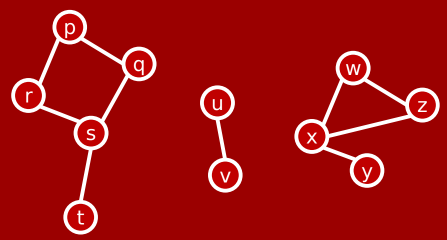
A **cycle** is a walk (of length at least 3) from u to u in which the only repeated vertex is u.

Example:
(p,r,s,q,p) is a cycle of length 4.





This 5-vertex graph is **connected**.



This 11-vertex graph is **not connected**.

It has 3 **connected components**:
{p,q,r,s,t}, {u,v}, {w,x,y,z}

Claim:
"is reachable from" is an *equivalence relation*

Proof:

- u is reachable from u? ✓
- u reachable from v
  $\Leftrightarrow$ v reachable from u? ✓
- u is reachable from v,
  v is reachable from w
  $\Rightarrow$ u is reachable from w? ✓

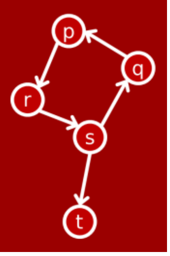
Connected components are the *equivalence classes*.

A little more about digraphs

In a digraph, walks have to "follow the arrows".

Given this, the reachable/walk/path/cycle stuff is all the same, except......

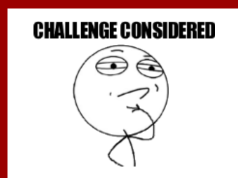
u reachable from v
$\not\Rightarrow$ v reachable from u



G is **strongly connected** iff
$\forall u,v \in V$, u is reachable from v.

Challenge:

Make an n-vertex graph connected using as few edges as possible.



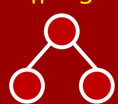
n = 1

Done
m = 0

n = 2

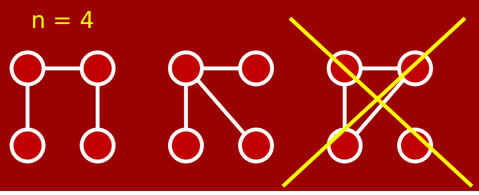
m = 1
necessary
and sufficient

n = 3



m = 2
necessary
and sufficient

n = 4



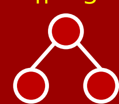
n = 1

Done
m = 0

n = 2

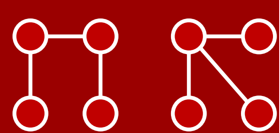
m = 1
necessary
and sufficient

n = 3



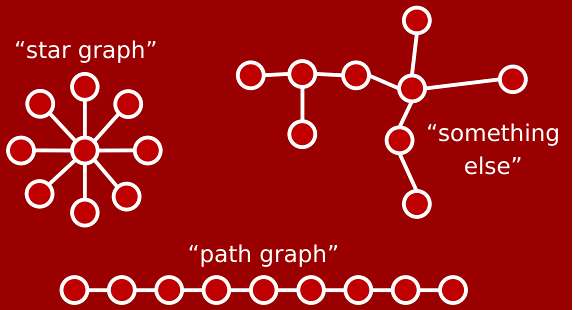
m = 2
necessary
and sufficient

n = 4



m = 3
necessary
and sufficient

$n-1$ edges are always **sufficient** to connect an n-vertex graph



$n-1$ edges are also **necessary** to connect an n-vertex graph

To prove this, we will use a lemma.

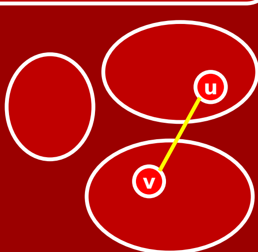
Lemma:
Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then G' has either k or $k-1$ connected components.

Lemma:
Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then G' has either k or $k-1$ connected components.

Example G with $k=3$ components:

Case 1: u,v in different components

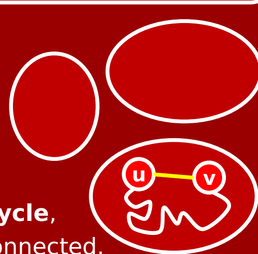
Then we go down to $k-1$ components.



Lemma:
Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then G' has either k or $k-1$ connected components.

Case 2: u,v in same component

Still have k components.

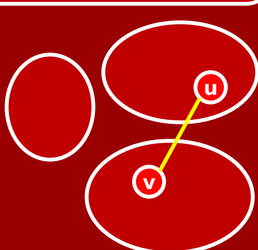


Bonus observation:
Adding $\{u,v\}$ creates a **cycle**, since u,v were already connected.

Lemma:
Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then G' has either k or $k-1$ connected components.

Case 1: u,v in different components

No cycle created, since it would have to involve u & v, but they weren’t previously connected.



Lemma:
Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then either:
a cycle was created, and G' has k components;
or no cycle was created, and G' has $k-1$ components.

Lemma: Let G be a graph with k connected components.
Let G' be formed by adding an edge between $u,v \in V$.
Then either: a cycle was created, and G' has k components;
or no cycle was created, and G' has $k-1$ components.

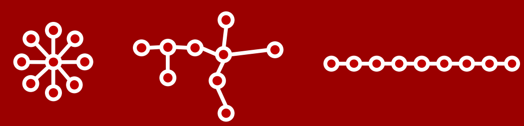
Theorem:
A connected n-vertex graph G has $\geq n-1$ edges.

Proof: Imagine adding in G's edges one by one.
Initially, n connected components.
Each edge can decrease # components by ≤ 1.
Have to get down to 1. Hence $\geq n-1$ edges. ☐

Bonus:
G has exactly $n-1$ edges iff it's **acyclic** (has no cycles).
Such a graph is called a **tree**.

Trees

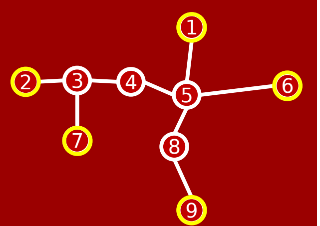
Example trees with $n = 9$ vertices.



Definition/Theorem:
An n-vertex **tree** is any graph with
at least 2 of the following 3 properties:
connected; $n-1$ edges; acyclic.
It will also automatically have the third.

Tree definitions

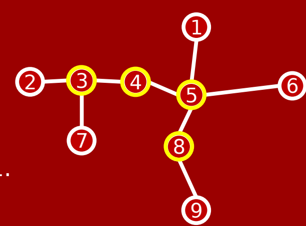
Leaf:
Vertex of degree 1.



Tree definitions

Leaf:
Vertex of degree 1.

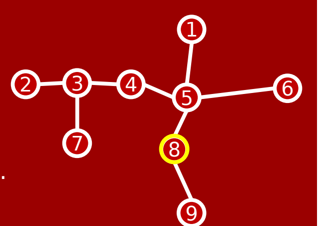
Internal node:
Vertex of degree > 1.



Tree definitions

Leaf:
Vertex of degree 1.

Internal node:
Vertex of degree > 1.

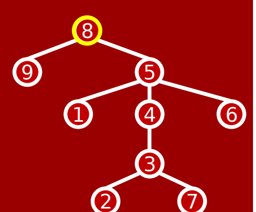


Rooted tree:
Tree with any one vertex designated as “root”.

Always drawn with root on top,
rest of tree “hanging down” from it.

Tree definitions

For rooted trees, we use
“family tree” terminology:
parent, **child**, **sibling**,
ancestor, **descendant**, etc.

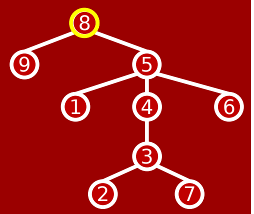


Rooted tree:
Tree with any one vertex designated as “root”.

Always drawn with root on top,
rest of tree “hanging down” from it.

Tree definitions

For rooted trees, we use "family tree" terminology: **parent**, **child**, **sibling**, **ancestor**, **descendant**, etc.



Binary tree: (cf. Lecture 2)
Rooted tree where each node has at most two children.

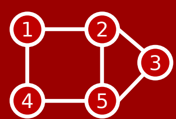
Time for some actual **computer science**.

Out of **all** computational problems in computer science, my personal favorite is…

Max-Cut

Max-Cut

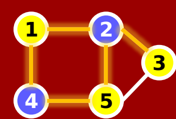
Input: A graph G=(V,E).



Output: A "**2-coloring**" of V:
each vertex designated yellow or blue.

Max-Cut

Input: A graph G=(V,E).



Output: A "**2-coloring**" of V:
each vertex designated yellow or blue.

Goal: Have as many **cut** edges as possible.
An edge is *cut* if its endpoints have different colors.

Motivation for Max-Cut

Say you're producing a TV show with n castaways.

You know the social network of friendships.

You need to split them into two tribes.

Naturally, as producer, you want to break up as many friendships as possible, to maximize drama-lama.

Motivation for Max-Cut

Motivating examples might be more natural if the social network recorded **enemyships**, rather than friendships.

There's an app for that.

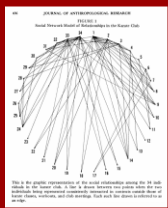
Enemybook

Kevin Matulef

"Enemybook is an antisocial utility that disconnects you with the people around you."

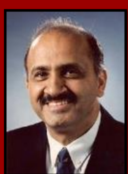
Motivation for Max-Cut

For example, given enemyship statuses for the Zachary Karate Club,



computing Max-Cut might give the best prediction for the schism into two clubs.

A “Local Search” Algorithm



Sartaj Sahni

Teofilo Gonzalez

1976

A “Local Search” Algorithm

Given input graph G with n vertices, m edges…

- Start with an arbitrary 2-coloring (say, all blue).
- Loop:
  - Check each vertex u to see if switching its color would increase the number of cut edges.
  - If such a vertex u is found, switch its color.
  - If no such vertex exists, halt.

Question: Why does this algorithm always halt?

Answer: After each loop iteration, # of cut edges increases by ≥ 1. Can’t go above m.

Corollary: Running time is $O((m+n)^2)$ (quadratic).

A “Local Search” Algorithm

Given input graph G with n vertices, m edges…

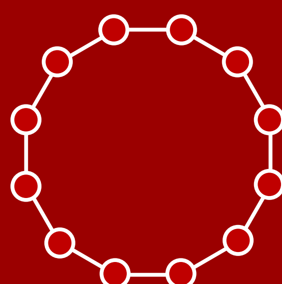
- Start with an arbitrary 2-coloring (say, all blue).
- Loop:
  - Check each vertex u to see if switching its color would increase the number of cut edges.
  - If such a vertex u is found, switch its color.
  - If no such vertex exists, halt.

Observation: In final 2-coloring, each vertex u has at least deg(u)/2 of its enemyships (edges) cut. (Why?)

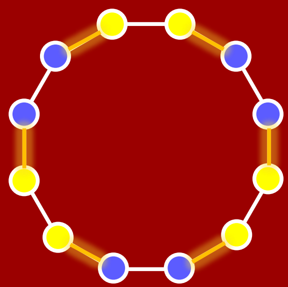
Conclusion:

Guaranteed to get $\geq \frac{m}{2}$ cut edges. (Exercise.)

This algorithm is pretty good. Is it optimal?



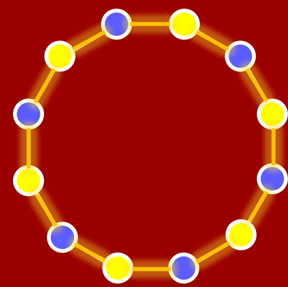
This algorithm is pretty good. Is it optimal?



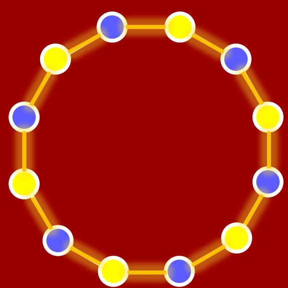
Maybe algorithms gets as far as this. $\frac{m}{2}$ edges cut

No single color switch gives any improvement.

This algorithm is pretty good. Is it optimal?



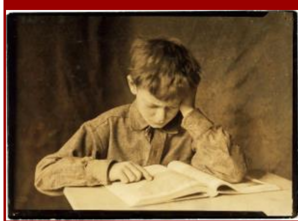
But the optimum 2-color cuts all m edges.



So is there a better polynomial-time algorithm?

Stay tuned…

Study Guide



Definitions:

Seriously, there were about 100 of them.

Theorems:

Sum of degrees = 2|E|.

The Theorem/Definition of trees.

Max-Cut local search analysis.