Notes 9

1 Graphs

Next kind of math structure: graph.

Extremely important in C.S.
Models many kinds of items and relationships among them.

E.g., networks and connectivity,
E.g., scheduling tasks with conflicting requirements

Lots of non-CS applications too.
E.g., cities and airline flights
E.g., atoms and molecular bonds

1.1 Definitions and Basic Examples

Graphs come in 2 flavors: *undirected* and *directed*.

1.1.1 Undirected Graphs

$G = (V, E)$, *vertices* and *edges*.
Vertices are just primitives – elements of some arbitrary set V.
Edges are unordered pairs (u, v) of vertices, $u \neq v$.

Example:

- $V = \{1, 2, 3, 4\}$

- $E = \{(1,2),(1,3),(1,4)(2,4)\}$

Give diagram.

Two important variants:

1. Sometimes undirected graphs are also allowed to have edges of the form (u, u). Loops (draw some).
2. Multiple edges (draw some).

Most often, just work with *simple* graphs, which have neither.

Now more examples.
Special kinds of graphs, also applications.

Example:
K_n, *complete graph* on n nodes.
Draw K_4.
Can be communication network graph, either physical or logical.
Write distributed program assuming complete connectivity.
May need lower-level translation to actual topology

Example:
C_n, cycle on n nodes
Correspond to ring networks

Example:
Star
For centralized communication

Example:
Mesh
City blocks
Processor connectivity in certain kinds of parallel computers

Example:

Hypercube
$V = \{0,1\}^n$, length n strings of bits
Again, connectivity in some parallel computers

Example:
Bipartite graph
"Two-part"
Vertices can be partitioned into two sets, and all edges connect elements of the two sets.
E.g., marriage relation represented by bipartite graph.
$K_{m,n}$ is *complete bipartite graph* for m vs. n nodes.

Example:
Another scheduling problem – different from the task scheduling problem from lecture 8.

Given n exams and set of pairwise *conflicts* (representing a student that has to take both exams), schedule the exams into t time slots without any conflicts.
That is, two conflicting exams cannot be given at the same time.

Represent problem by *conflict graph*.
Nodes represent exams, two nodes joined by edge if corresponding exams cannot be held at same time. Graph representation:

Figure 1: Conflict graph

- $V = exams$
- $E = \{(u,v)|u$ and v conflict $\}$
- $t = 3$, say Wednesday 5-7, 7-9, 9-11

Will return to this example later in the hour with some ideas on how to solve.

Terminology:

u, v *adjacent*, *neighbors*
$e = (u, v)$ *incident* with vertices u, v
u and v *endpoints* of e

degree of vertex = number of edges incident with it
$deg(v)$

1.1.2 Directed Graphs

Again, $G = (V, E)$, *vertices* and *edges*.
Now edges are *ordered* pairs (u, v) of vertices, possibly with $u = v$ (allowing loops).

Example:

- V = 1,2,3,4
- E = (1,2), (1,3), (1,4) (2,4)

Give diagram – now arrows, with direction.

Again, variant allows multiple edges.
Notice that the default case is different for undirected and directed graphs.
Undirected: no loops.
Directed: loops.
Neither case has multiple edges as default, though.

Can view undirected graph as special case of directed (with no loops, and with two directed edges representing every undirected edge).

Example:
Cities with airline connections
No loops here.

Example:
Directed cycle
Communication network (e.g., token ring)

Example:
Task precedence graph, from lecture 8

Example:
Round-robin tournament
Complete graph, one direction for each edge indicating who defeated whom.
Could be cycles, though.

Terminology:

$e = (u, v)$ has u as *initial vertex* and v as *terminal vertex*.
(or something like that)
v *out-neighbor* of u, u *in-neighbor* of v

in-degree, number of in-neighbors,
out-degree, number of out-neighbors

If (u, u) is an edge, then u is both an in-neighbor and an out-neighbor of itself, and contributes 1 to both the in-degree and the out-degree.

1.2 Simple Theorems about Degrees

1.2.1 Undirected

Theorem 1.1 (Handshaking Theorem)
The sum of the degrees of all the nodes is equal to twice the number of edges.
$\Sigma_{v \in V} deg(v) = 2|E|$.

Proof. Each edge adds 1 to the degree of each of its endpoints.

If that's not convincing as is, think of this as the inductive step in an inductive proof.
Base: No edges – both sides of equation are 0.
Inductive: Add 2 to RHS and to LHS.

∎

1.2.2 Directed

Analogous:

Theorem 1.2 The sum of the in-degrees of all nodes is equal to the sum of the out-degrees of all nodes, which is equal to the number of edges.
$\Sigma_{v \in V} \textit{in-degree}(v) = \Sigma_{v \in V} \textit{out-degree}(v) = |E|$.

Similar proof to the undirected case.
Notice that it's important that we count loop edge in in-degree and out-degree of its endpoint.

1.3 Graph Representation

Adjacency lists vs. adjacency matrices

1.3.1 Adjacency Lists

For each vertex, list all its neighbors (undirected graph), or its out-neighbors (directed graph).

Example: First example above, undirected

- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (1, 3), (1, 4)(2, 4)\}$

Represent as adjacency lists:
$(1, (2, 3, 4)), (2, (1, 4)), (3, (1)), (4, (1, 2))$

Example: First example, directed
$(1, (2, 3, 4)), (2, (4)), (3, ()), (4, ())$

1.3.2 Adjacency Matrices

Undirected:
$(0, 1)$-Matrix with rows and columns corresponding to V.
1 in position (u, v) if (u, v) is in E.
Notice this means that (v, u) is also in E, since undirected.
So there's also a 1 in position (v, u).

Symmetric matrix, symmetric around major diagonal.
Major diagonal empty, since no self-loops.

Example: Same one as above

```
1: 0 1 1 1
2: 1 0 0 1
3: 1 0 0 0
4: 1 1 0 0
```

Directed:
Again, matrix with rows and columns corresponding to V.
1 in position (u, v) if (u, v) is in E.
Now needn't be symmetric.
May have some 1 entries on major diagonal.

Example: From above

```
1: 0 1 1 1
2: 0 0 0 1
3: 0 0 0 0
4: 0 0 0 0
```

1.4 Paths and Connectivity

Many of the basic results about graphs have to do with paths from one node to another.

1.4.1 Undirected Graphs, Basic defs and results

Path is a sequence of edges that match up, end to end.
Formally, a sequence $e_1 = (u_1, v_1), e_2 = (u_2, v_2), \ldots, e_k = (u_k, v_k)$ such that for all i, $1 \leq i \leq k-1$, $v_i = u_{i+1}$.

$k =$ path *length*.
(Trivial case: path of length 0)

Path is *simple* if it has no repeated edges.

Path is a *circuit* if it begins and ends at the same vertex, $u_1 = v_k$.

Undirected graph is *connected* if there is a path between every pair of vertices.

Theorem 1.3 If in an undirected graph, there is a path from u to v, then there is a simple path from u to v (in fact, one with no repeated vertices).

Proof. Consider a shortest length path.
(Must exist, by well-ordering.)
If it's not simple, then there's a vertex x that repeats.
Splice out the middle part, getting a shorter path.

■

1.4.2 Directed Graphs, Basic defs and results

Basically the same defs as for undirected.

Path is a sequence of (directed) edges that match up, end to end.
Formally, a sequence $e_1 = (u_1, v_1), e_2 = (u_2, v_2), \ldots, e_k = (u_k, v_k)$ such that for all i, $1 \leq i \leq k-1$, $v_i = u_{i+1}$.

$k =$ path *length*.
(Trivial case: path of length 0)

Path is *simple* if it has no repeated edges.

Path is a *circuit* if it begins and ends at the same vertex, $u_1 = v_k$.

Directed graph is *strongly connected* if there is a path between every pair of vertices.
Sometimes also talk about a digraph being *weakly connected*, if there's a connection by following edges either forwards or backwards, indiscriminately.
Formally, "underlying undirected graph" (obtained by deleting loops and undirecting all edges) is connected.

Theorem 1.4 If in a directed graph, there is a path from u to v, then there is a simple path from u to v (in fact, one with no repeated vertices – rules out use of loops too).

Proof. Same proof as for the undirected case. ■

1.4.3 Paths and Adjacency Matrices

Adjacency matrices are especially convenient for giving information about paths and connectivity.

Example:
Consider a new example digraph, given by adjacency matrix:

```
1: 1 1 0 0 0
2: 0 1 1 0 1
3: 0 0 0 1 1
4: 1 0 0 0 0
5: 0 0 0 0 0
```

Draw graph.
Also think of this graph as representing a binary relation R on V.

Recall when we studied relations, we defined powers of relations (based on products).
E.g., consider $R \times R = R^2$.
Recall definition: $(u, v) \in R \times R$ exactly if there exists x such that $(u, x) \in R$ and $(x, v) \in R$.
Here, the matrix for $R \times R$ is:

```
1: 1 1 1 0 1
2: 0 1 1 1 1
3: 1 0 0 0 0
4: 1 1 0 0 0
5: 0 0 0 0 0
```

Explain how entries are filled in.
E.g., entry $(4, 2)$ comes about because $(4, 1)$ and $(1, 2)$ are in R.
In general, get entry for (u, v) to be $\bigvee_{x \in V}((u, x) \in R \wedge (x, v) \in R)$.

Notice similarity with matrix multiplication.
Using Boolean or in place of $+$ and Boolean and in place of $\times$.

Q: What does this new relation/matrix mean in terms of the original graph?
A: Has 1 in position (u, v) exactly if there is a path of length exactly 2 from u to v in the graph.
(Check this.)

Q: What if we did R^3, cubing the matrix instead of squaring?
A: Would give existence of length 3.

In general, taking R^n, which is raising the matrix to the nth power, gives existence of length n paths.

Now suppose, for fun, that we multiplied the matrices in the *ordinary* way, using + and $\times$.
E.g., for the squaring case:

```
1: 1 2 1 0 1
2: 0 1 1 1 2
3: 1 0 0 0 0
4: 1 1 0 0 0
5: 0 0 0 0 0
```

Go over entries.
E.g., for the $(2, 5)$ entry, can have either 2 or 3 as intermediate vertex.

Q: What do the numbers mean?
A: Number of *distinct* paths of length exactly 2 from u to v.
Distinct here doesn't mean disjoint – can have some edges in common.
Just can't be all the same.

In general, the square of the adjacency matrix of a digraph tells you how many paths of length 2 there are between all pairs of nodes.

In fact, for any k, the kth power of the adjacency matrix yields the number of paths of length exactly k between all pairs of nodes.
The (u, v) entry says how many from u to v.

Theorem 1.5 Suppose that $G = (V, E)$ is any directed graph and A is its adjacency matrix. Then for any nodes $u, v \in V$, the (u, v) entry of A^k is the number of distinct paths of length exactly k from u to v in G.

Proof. By induction on k. Fix G.
Define $P(k)$ to be:
For all pairs of nodes $u, v \in V$, the (u, v) entry of A^k is the number of distinct paths of length exactly k from u to v in G.

Base: $P(1)$. $A^1 = A$. This is the number of paths of length 1, by definition of adjacency matrix.

Inductive step:
Assume $P(k-1)$, show $P(k)$.
Fix u, v, consider (u, v) element of A^k.
Write this as $a^{(k)}_{u,v}$.

Note that $A^k = A^{k-1} \cdot A$.
Therefore, by the definition of matrix multiplication, we have:

$$a_{u,v}^{(k)} = \sum_{x \in V} a_{u,x}^{(k-1)} a_{x,v}.$$

Note that $a_{x,v} = 1$ if and only if $(x, v) \in E$, and otherwise it is 0.
This is by definition of the adjacency matrix A.

Also, by inductive hypothesis $P(k-1)$, $a_{u,x}^{(k-1)}$ is equal to the number of distinct paths in G from u to x of length exactly $k-1$.
Therefore, the product $a_{u,x}^{(k-1)} a_{x,v}$ is equal to the number of paths from u to v of length k whose next-to-last node is x.
Thus, the *total number* of paths from u to v of length exactly k is obtained by taking the sum of these products for all x, which gives the sum we claimed. ∎

This was all described for digraphs.
But it works for undirected graphs too, since they are a special case of digraphs (no loops, edges in both directions).

Theorem 1.6 Suppose that $G = (V, E)$ is any undirected graph and A is its adjacency matrix.
Then for any nodes $u, v \in V$, the (u, v) entry of A^k is the number of distinct paths of length exactly k from u to v in G.

Theorem follows as a consequence of the theorem for digraphs.

Example:
Consider new example graph, given by:

```
1: 0 1 0 1 0
2: 1 0 1 0 1
3: 0 1 0 1 1
4: 1 0 1 0 0
5: 0 1 1 0 0
```

Draw graph.
Then A^2 is:

```
1: 2 0 2 0 1
2: 0 3 1 2 1
```

```
3: 2 1 3 0 1
4: 0 2 0 2 1
5: 1 1 1 1 2
```

Gives the number of length 2 paths.

1.4.4 Reflexive, Transitive Closure(s)

Recall for posets, defined notions of *reflexive closure*, *transitive closure* and *reflexive transitive closure.*
Amounts to adding in some additional ordering edges.
Same ideas make sense for an arbitrary digraph.

Reflexive closure: Add in all edges (u, u).

Transitive closure: Add in all edges (u, v) such that (u, x) and (x, v) are in G. Repeat till no new edges.
Or, alternatively, add in any edge (u_1, u_k) for which all the edges $(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k)$ are in G.

Reflexive transitive closure: Add in both.

Q: What does the reflexive closure of a graph G represent?
A: Has an edge (u, v) exactly if can reach from u to v in 1 or 0 steps in G.

Q: What does the transitive closure represent?
A: Edge (u, v) exactly if can reach from u to v in 1 or more steps.

Q: Reflexive transitive closure?
A: Edge (u, v) exactly if can reach from u to v at all (in any number, 0 or more, steps).

Given adjacency matrix A of graph G, how to compute adjacency matrices of the closures?
The reflexive part is easy – just add in the diagonal entries.
Transitive?

Algorithm for transitive closure:
1. Compute $A^1, A^2, A^3, \ldots, A^n$.
2. Take the sum of all of them.
3. Replace all nonzero entries by 1, keep the 0 entries.

Equivalently do Boolean and/or matrix multiplications, take the “or” instead of sum.

Q: Why only need to consider up to n?
A: If there's a path, there's one of length at most n (with no repeated vertices).

Algorithm for reflexive transitive closure is easier:
1. Add in all major diagonal entries, yielding B.
2. Compute B^n
2. Replace all nonzero entries by 1, keep 0 entries.

Equivalently, do B^n using Boolean matrix ops.

Q: Why is just B^n sufficient here?
A: If there's a path of length $\leq n$, then there's a path of length exactly n, using self-loops.

Pretty efficient.
In fact, can use fast multiplication tricks, based on successive squaring, to cut down the number of matrix multiplications to about $2 \log n$ rather than $n - 1$:

$Fastexp(B, n)$, $n \geq 1$:
    if $n = 1$ then return B
    else if n is even then
        $C := Fastexp(B, \frac{n}{2})$
        return C^2
    else
        $C := Fastexp(B, n - 1)$
        return $C \times B$

Associativity of matrix multiplication implies that this works OK.

Example:

```
1: 1 1 0 0 0
2: 0 1 1 0 1
3: 0 0 0 1 1
4: 1 0 0 0 0
5: 0 0 0 0 0
```

Reflexive transitive closure is:

```
1: 1 1 1 1 1
```

```
2: 1 1 1 1 1
3: 1 1 1 1 1
4: 1 1 1 1 1
5: 0 0 0 0 1
```

Says which nodes can be reached from which others.

Algorithm for strong connectivity of digraph:
Start with digraph adjacency matrix A.
Compute reflexive transitive closure matrix B.
See whether or not all entries of B are equal to 1.

Undirected graph?
Useful to transform to digraph, via bidirectional edges.
Then can compute reflexive transitive closure.
What info do we get from this?
Get 1 in position (u, v) exactly if there's a path from u to v in the original undirected graph.

Determine connectivity of entire graph by seeing whether all entries are equal to 1.
Note: The graph represented by the all-1's adjacency matrix is the complete (undirected) graph.

Q: If the graph is not connected, then what do the final adjacency matrix and final graph look like?
A: Graph consists of a collection of disjoint complete graphs.
Matrix has 1's in "blocks".

The set of vertices in any one of these disjoint pieces is called a *connected component*.
Two nodes u and v are in the same connected component of a graph G if there is a path between u and v.

I'll leave you to figure out what the r-t closure adjacency matrix and graph look like for non-strongly-connected digraph.

Example: Digraph reachability, non-strongly-connected graph, components
Vertices are integers from 1 to 24
Edge (u, v) provided that:

1. Every prime that divides u also divides v.
   (Different numbers of times allowed, though.)

2. Either $v = pu$ or $u = vp$, for some prime p.

Sample edges:

```
$1: 2,3,5,7,11,13,17,19,23$ \\
$2: 4,6,10,14,22$ \\
$3: 6,9,15,21$ \\
$4: 2,8,12,20$ \\
$5: 10,15$ \\
$6: 12,18$ \\
$7: 14,21$ \\
$8: 4,16,24$ \\
$9: 3,18$ \\
$10: 20$ \\
$11: 22$ \\
$12: 6,24$ \\
$13:$ \\
$14:$ \\
$15:$ \\
$16: 8$ \\
$17:$ \\
$18: 6$ \\
$19:$ \\
$20: 10$ \\
$21:$ \\
$22:$ \\
$23:$ \\
$24: 12$ \\
```

Strongly connected components: Those with same prime factors:
$(2, 4, 8, 16)$, $(3, 9)$, $(6, 12, 18, 24)$, $(10, 20)$, others singletons.

Example:
Digraph for roads, can only drive one way, not necessarily strongly connected.
SCC consists of cities s.t. can drive between any pair.

1.5 Graph Coloring

Back to conflict graph example.
Undirected conflict graph (no loops, recall).

Want to assign one of t time slots to each exam s.t. no two with same time slot conflict.
Think of the time slots as "colors", blue, green, etc.
The graph problem is to assign colors to the vertices of the graph in such a way that the two endpoints of any edge are colored differently.
Graph coloring

In general, graph coloring is a very difficult problem.
It's a classic example of a problem for which no fast optimal algorithms are known.
However, in some cases, can get good upper bounds.

Theorem 1.7 If the degree of every node in G is $\leq d$, then G can be colored with $\leq d+1$ colors.

Proof. By induction on the number n of nodes. Fix d.
$P(n)$: Any n-vertex graph with node degrees $\leq d$ is $d+1$-colorable.

Base: $P(1)$ is true because 1-node graph is 1-colorable, hence $d+1$-colorable.

Inductive step:
Show for all n, $P(n) \Rightarrow P(n+1)$.
Let G be any $n+1$-node graph with node degrees $\leq d$.
Select any node u and remove it to create $G' = G - u$.
(Also removes all the edges incident on u.)

Then G' has n nodes and maximum degree $\leq d$.
By inductive hypothesis, G' is $(d+1)$-colorable.
To complete the proof, all we need to do is find a color for u.
This is easy, since the degree of u is at most d, so there is some color not used for the neighbors of u.
Thus, G is $(d+1)$-colorable, so $P(n+1)$ holds and the proof is complete. ∎

For some graphs with degree d, $(d+1)$ colors is the best we can do.
E.g., bound is optimal for the complete graph K_{d+1}, which has degree d.
All its nodes must have different colors, because they are all connected.

In some other cases, the bound is very far from optimum.
E.g., star graph with $d+1$ nodes has max degree d but can be colored with 2 colors.

Example: Ring
Even: 2
Odd: 3

Example: Bipartite graphs
Always 2-colorable

Applications:

Example:
Mentioned conflict scheduling.
Used in operating systems, to schedule tasks with conflicting resource requirements.
Schedule conflicting programs at different times.

Example: (Chemicals)
If you have several chemicals to store, you may want to put jars with incompatible chemicals on different shelves.
You want to know how many shelves you will need.
In this case, the nodes are the jars, edges the incompatibility relationships, colors the shelves.

Example: Index registers
Compilers keep the values of some frequently used program variables in very fast cache (index registers)
Certain variables need to coexist in registers – those used at the same time during execution.
Say these conflict.
So can't be stored in the same register.
Want to assign register numbers to variables in such a way that "conflicting" variables not assigned to same register.
Coloring variables with register (numbers)

Example: Map coloring
Nodes represent countries, edges drawn between nodes representing adjacent countrires.
This yields graphs of very special kind, called *planar* – can be drawn in a plane, with no lines crossing.
It turns out that planar graphs are colorable using a small number of colors – only 4.

Proof uses computers.
Hand proof first says that if there are any non-4-colorable planar graphs, then there has to be a counterexample among a certain (large - about 2000) collection of graphs.
Then computer program shows that all of these 2000 graphs are 4-colorable.

At the time this proof was discovered, around 1976, the math community debated whether it was even a proof (because all the computer cases meant that it was just too long for any person to understand it – how to become convinced it was correct?)
Now I think such proofs are generally accepted, but subject to confidence in correctness of the program.

1.6 Recitation: Euler and Hamiltonian Circuits (and Paths)

Nice material for recitation in Sections 7.5-7.7.

1. Define Euler circuit.
Do the characterization theorem, Theorem 1, p. 476

2. Define Euler path.
Characterization theorem, Theorem 2, p. 478.

3. Define Hamiltoniam path and circuit, say the problem is hard in general.
Do a few examples.

Euler's Theorem, p. 502 is nice.

1.7 Tutorial Problems

1. Posets

2. Graphs Conflicts and coloring.

Consider a collection of tasks $T_1, ...T_n$, where each T_i has a set of resource requirements S_i. (Each S_i is a subset of R, some universal set of resources.) Each task needs exclusive access to its resources when it executes. Each task takes time 1 to execute.

(a) In terms of the T_i's and the S_i's, define a conflict graph describing which tasks cannot execute at the same time.

(b) If each task is supposed to execute once, describe a minimum-time schedule for all of them to execute. Describe this in terms of your conflict graph. How long does it take for all the tasks to execute?

(c) Now suppose that each task is supposed to execute an arbitrary numbers of times. Specifically, there is a *scheduler* who interacts with a *client*. Initially, the client submits some subset of the tasks that it wants to have performed. The scheduler responds by scheduling some of these to execute immediately, and leaves the others *pending*. At each successive time step, the client submits an additional set of tasks; these cannot include any pending tasks, but can include tasks that have already been performed at previous time steps.

You are to design a strategy for the scheduler to ensure that all requested tasks eventually get performed. Try to minimize the worst-case waiting time for an individual task.

2. Again consider a set of tasks $T_1, \ldots, T_n$, where each has a set of resource requirements S_i. Again, each tasks needs exclusive access to its resources, and each task takes time 1. But now, there is also a precedence relation between the tasks – some tasks are required to precede other tasks.

Develop a good method for scheduling all the tasks, each exactly once. Try to minimize the total schedule time.