Breadth-First Search

Input $G(V, E)$ [a connected graph]
v [start vertex]

Algorithm Breadth-First Search

visit v
$V' \leftarrow \{v\}$ [V' is the vertices already visited]
Put v on Q [Q is a queue]
repeat while $Q \neq \emptyset$
  $u \leftarrow head(Q)$ [$head(Q)$ is the first item on Q]
  for $w \in A(u)$ [$A(u) = \{w | \{u, w\} \in E\}$]
    if $w \notin V'$
    then visit w
      Put w on Q
      $V' \leftarrow V' \cup \{w\}$
    endif
  endfor
  Delete u from Q

The BFS algorithm basically finds a tree embedded in the graph.

- This is called the *BFS search tree*

BFS and Shortest Length Paths

If all edges have equal length, we can extend this algorithm to find the shortest path length from v to any other vertex:

- Store the path length with each node when you add it.
- Length(v) = 0.
- Length(w) = Length(u) + 1

With a little more work, can actually output the shortest path from u to v.

- This is an example of how BFS and DFS arise unexpectedly in a number of applications.
  - We'll see a few more

Depth-First Search

Input $G(V, E)$ [a connected graph]
v [start vertex]

Algorithm Depth-First Search

visit v
$V' \leftarrow \{v\}$ [V' is the vertices already visited]
Put v on S [S is a stack]
$u \leftarrow v$
repeat while $S \neq \emptyset$
if $A(u) - V' \neq \emptyset$
then Choose $w \in A(u) - V'$
  visit w
  $V' = V' \cup \{w\}$
  Put w on stack
  $u \leftarrow w$
else $u \leftarrow top(S)$ [Pop the stack]
endif
endrepeat

DFS uses *backtracking*

- Go as far as you can until you get stuck
- Then go back to the first point you had an untried choice

Spanning Trees

A *spanning tree* of a connected graph $G(V, E)$ is a connected acyclic subgraph of G, which includes all the vertices in V and only (some) edges from E.

Think of a spanning tree as a "backbone"; a minimal set of edges that will let you get everywhere in a graph.

- Technically, a spanning tree isn't a tree, because it isn't directed.

The BFS search tree and the DFS search tree are both spanning trees.

- In the text, they give algorithms to produce minimum weight spanning trees
- That's done in CS 482, so we won't do it here.

Graph Coloring

How many colors do you need to color the vertices of a graph so that no two adjacent vertices have the same color?

- Application: scheduling
  - Vertices of the graph are courses
  - Two courses taught by same prof are joined by edge
  - Colors are possible times class can be taught.

Lots of similar applications:

- E.g. assigning wavelengths to cell phone conversations to avoid interference.
  - Vertices are conversations
  - Edges between "nearby" conversations
  - Colors are wavelengths.
- Scheduling final exams
  - Vertices are courses
  - Edges between courses with overlapping enrollment
  - Colors are exam times.

Chromatic Number

The *chromatic number* of a graph G, written $\chi(G)$, is the smallest number of colors needed to color it so that no two adjacent vertices have the same color.

Examples:

A graph G is *k-colorable* if $k \geq \chi(G)$.

Determining $\chi(G)$

Some observations:

- If G is a complete graph with n vertices, $\chi(G) = n$
- If G has a clique of size k, then $\chi(G) \geq k$.
  - Let $c(G)$ be the *clique number* of G: the size of the largest clique in G. Then
  $$\chi(G) \geq c(G)$$
- If $\Delta(G)$ is the maximum degree of any vertex, then
  $$\chi(G) \leq \Delta(G) + 1 :$$
  - Color G one vertex at a time; color each vertex with the "smallest" color not used for a colored vertex adjacent to it.

How hard is it to determine if $\chi(G) \leq k$?

- It's NP complete, just like
  - determining if $c(G) \geq k$
  - determining if G has a Hamiltonian path
  - determining if a propositional formula is satisfiable

Can guess and verify.

Bipartite Graphs

A graph $G(V, E)$ is *bipartite* if we can partition V into disjoint sets V_1 and V_2 such that all the edges in E joins a vertex in V_1 to one in V_2.

- A graph is bipartite iff it is 2-colorable
- Everything in V_1 gets one color, everything in V_2 gets the other color.

Example: Suppose we want to represent the "is or has been married to" relation on people. Can partition the set V of people into males (V_1) and females (V_2). Edges join two people who are or have been married.

Characterizing Bipartite Graphs

Theorem: G is bipartite iff G has no odd-length cycles.

Proof: Suppose that G is bipartite, and it has edges only between V_1 and V_2. Suppose, to get a contradiction, that $(x_0, x_1, \ldots, x_{2k}, x_0)$ is an odd-length cycle. If $x_0 \in V_1$, then x_2 is in V_1. An easy induction argument shows that $x_{2i} \in V_1$ and $x_{2i+1} \in V_2$ for $0 = 1, \ldots, k$. But then the edge between x_{2k} and x_0 means that there is an edge between two nodes in V_1; this is a contradiction.

- Get a similar contradiction if $x_0 \in V_2$.

Conversely, suppose $G(V, E)$ has no odd-length cycles.

- Partition the vertices in V into two sets as follows:
  - Start at an arbitrary vertex x_0; put it in V_0.
  - Put all the vertices one step from x_0 into V_1
  - Put all the vertices two steps from x_0 into V_0;
  - ...

This construction works if G is connected and has no odd-length cycles.

- What if G isn't connected?

This construction also gives a polynomial-time algorithm for checking if a graph is bipartite.

The Four-Color Theorem

Can a map be colored with four colors, so that no countries with common borders have the same color?

- This is an instance of graph coloring
  - The vertices are countries
  - Two vertices are joined by an edge if the countries they represent have a common border

A *planar graph* is one where all the edges can be drawn on a plane (piece of paper) without any edges crossing.

- The graph of a map is planar

Graphs that are planar and ones that aren't:

Four-Color Theorem: Every map can be colored using at most four colors so that no two countries with a common boundary have the same color.

- Equivalently: every planar graph is four-colorable

Four-Color Theorem: History

- First conjectured by in 1852
- Five-colorability was long known
- "Proof" given in 1879; proof shown wrong in 1891
- Proved by Appel and Haken in 1976
  - 140 journal pages + 100 hours of computer time
  - They reduced it to 1936 cases, which they checked by computer
- Proof simplified in 1996 by Robertson, Sanders, Seymour, and Thomas
  - But even their proof requires computer checking
  - They also gave an $O(n^2)$ algorithm for four coloring a planar graph
- Proof checked by Coq theorem prover (Werner and Gonthier) in 2004
  - So you don't have to trust the proof, just the theorem prover

Note that the theorem doesn't apply to countries with non-contiguous regions (like U.S. and Alaska).

Topological Sorting

[NOT IN TEXT]

If $G(V, E)$ is a *dag*: directed acyclic graph, then a *topological sort* of G is a total ordering $\prec$ of the vertices in V such that if $(v, v') \in E$, then $v \prec v'$.

- Application: suppose we want to schedule jobs, but some jobs have to be done before others
  - vertices on dag represent jobs
  - edges describe precedence
  - topological sort gives an acceptable schedule

Theorem: Every dag has at least one topological sort.

Proof: Two algorithms. Both depend on this fact:

- If $V \neq \emptyset$, some vertices in V have indegree 0.
  - If all vertices in V have indegree > 0, then G has a cycle: start at some $v \in V$, go to a parent v' of v, a parent v'' of v', etc.
    - Eventually a node is repeated; this gives a cycle

Algorithm 1: Number the nodes of indegree 0 arbitrarily. Then remove them and the edges leading out of them. You still have a dag. It has nodes of indegree 0. Number them arbitrarily (but with a higer number than the original set of nodes of indegree 0). Continue ... This gives a topological sort.

Algorithm 2: Add a "virtual node" v^* to the graph, and an edge from v^* to all nodes with indegree 0

- Do a DFS starting at v^*. Output a node after you've processed all the children of that node.
  - Note that you'll output v^* last
  - If there's an edge from u to v, you'll output v before u
- Reverse the order (so that v^* is first) and drop v^*

That's a topological sort.

- This can be done in time linear in $|V| + |E|$

Graph Isomorphism

When are two graphs that may look different when they're drawn, really the same?

Answer: $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are *isomorphic* if they have the same number of vertices ($|V_1| = |V_2|$) and we can relabel the vertices in G_2 so that the edge sets are identical.

- Formally, G_1 is isomorphic to G_2 if there is a bijection $f : V_1 \to V_2$ such that $\{v, v'\} \in E_1$ iff $(\{f(v), f(v')\} \in E_2$.
- Note this means that $|E_1| = |E_2|$

Checking for Graph Isomorphism

There are some obvious requirements for $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ to be isomorphic:

- $|V_1| = |V_2|$
- $|E_1| = |E_2|$
- for each d, #(vertices in V_1 with degree d) = #(vertices in V_1 with degree d)

Checking for isomorphism is in NP:

- Guess an isomorphism f and verify
- We believe it's not in polynomial time and not NP complete.

Game Trees

Trees are particularly useful for representing and analyzing games.

Example *Daisy* (aka *Nim*):

- players alternate picking petals from a daisy.
- A player gets to pick 1 or 2 petals.
- Whoever picks the last one wins.
- There's another version where whoever takes the last one loses
  - both get analyzed the same way

Here's the game tree for 4-petal daisy:

A Fun Application of Graphs

A farmer is bringing a wolf, a cabbage, and a goat to market. They need to cross a river in a boat which can accommodate only two things, including the farmer. Moreover:

- the farmer can't leave the wolf alone with the goat
- the farmer can't leave the goat alone with the cabbage

How should he cross the river?

Getting a good representation is the key.

What are the allowable configurations?

- A configuration looks like (X, Y), where $X, Y \subseteq \{W, C, F, G\}$, $Y = \overline{X}$
- Can have X on the initial side of the river, Y on the other

$(WCFG, \emptyset)$	$(\emptyset, WCFG)$
(WCF, G)	(G, WCF)
(WGF, C)	(C, WGF)
(CGF, W)	(FG, WC)
(WC, FG)	(W, CFG)

- Disallowed configurations: (WG, FC), (GC, FW), (FC, WG), (FW, GC)
- Initial configuration: $(WCFG, \emptyset)$.

Use a graph to represent when we can get from one configuration to another.

Some Bureuacracy

- The final is on Thursday, May 8, 7-9:30 PM, in UP B17
- If you have a conflict and haven't told me, let me know now right away
  - Also tell me the courses and professors involved (with emails)
  - Also tell the other professors
  - We may schedule a makeup; or perhaps the other course will.
- Office hours go on as usual during study week, but check the course web site soon.
  - There may be small changes to accommodate the TA's exams
- There will be a review session

Coverage of Final

- everything covered by the first prelim
  - emphasis on more recent material
- Chapter 4: Fundamental Counting Methods
  - Permutations and combinations
  - Combinatorial identities
  - Pascal's triangle
  - Binomial Theorem (but not multinomial theorem)
  - Balls and urns
  - Inclusion-exclusion
  - Pigeonhole principle
- Chapter 6: Probability:
  - 6.1–6.5 (but not inverse binomial distribution)
  - basic definitions: probability space, events
  - conditional probability, independence, Bayes Thm.
  - random variables
  - uniform, binomial, and Poisson distributions
  - expected value and variance
  - Markov + Chebyshev inequalities

- Chapter 7: Logic:
  - 7.1–7.4, 7.6, 7.7; *not* 7.5
  - translating from English to propositional (or first-order) logic
  - truth tables and axiomatic proofs
  - algorithm verification
  - first-order logic
- Chapter 3: Graphs and Teres
  - basic terminology: digraph, dag, degree, multigraph, path, connected component, clique
  - Eulerian and Hamiltonian paths
    - algorithm for telling if graph has Eulerian path
  - BFS and DFS
  - bipartite graphs
  - graph coloring and chromatic number
  - topological sort
  - graph isomorphism

Ten Powerful Ideas

- **Counting**: Count without counting (*combinatorics*)
- **Induction**: Recognize it in all its guises.
- **Exemplification**: Find a sense in which you can try out a problem or solution on small examples.
- **Abstraction**: Abstract away the inessential features of a problem.
  - One possible way: represent it as a graph
- **Modularity**: Decompose a complex problem into simpler subproblems.
- **Representation**: Understand the relationships between different possible representations of the same information or idea.
  - Graphs vs. matrices vs. relations
- **Refinement**: The best solutions come from a process of repeatedly refining and inventing alternative solutions.
- **Toolbox**: Build up your vocabulary of abstract structures.

- **Optimization**: Understand which improvements are worth it.
- **Probabilistic methods**: Flipping a coin can be surprisingly helpful!

Connections: Random Graphs

Suppose we have a random graph with n vertices. How likely is it to be connected?

- What is a *random* graph?
  - If it has n vertices, there are $C(n,2)$ possible edges, and $2^{C(n,2)}$ possible graphs. What fraction of them is connected?
  - One way of thinking about this. Build a graph using a random process, that puts each edge in with probability $1/2$.

- Given three vertices a, b, and c, what's the probability that there is an edge between a and b and between b and c? $1/4$
- What is the probability that there is no path of length 2 between a and c? $(3/4)^{n-2}$
- What is the probability that there is a path of length 2 between a and c? $1-(3/4)^{n-2}$
- What is the probability that there is a path of length 2 between a and every other vertex? $> (1-(3/4)^{n-2})^{n-1}$

Now use the binomial theorem to compute $(1-(3/4)^{n-2})^{n-1}$

$$\begin{aligned} & (1-(3/4)^{n-2})^{n-1} \\ = \ & 1-(n-1)(3/4)^{n-2}+C(n-1,2)(3/4)^{2(n-2)}+\cdots \end{aligned}$$

For sufficiently large n, this will be (just about) 1.

Bottom line: If n is large, then it is almost certain that a random graph will be connected.

Theorem: [Fagin, 1976] If P is *any* property expressible in first-order logic, it is either true in almost all graphs, or false in almost all graphs.

This is called a *0-1 law*.

Connection: First-order Logic

Suppose you wanted to query a database. How do you do it?

Modern database query language date back to SQL (structured query language), and are all based on first-order logic.

- The idea goes back to Ted Codd, who invented the notion of relational databases.

Suppose you're a travel agent and want to query the airline database about whether there are flights from Ithaca to Santa Fe.

- How are cities and flights between them represented?
- How do we form this query?

You're actually asking whether there is a path from Ithaca to Santa Fe in the graph.

- This fact cannot be expressed in first-order logic!