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 Galton (Darwin's cousin) in 1852
- False proofs given in 1879, 1880; disproved in 1891
- Computer proof given by Appel and Haken in 1976
  - 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).

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 $f : V_1 \rightarrow V_2$ 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.

Patterns and Finite Automata

A *pattern* is a set of objects with a recognizable property.

- In computer science, we're typically interested in patterns that are sequences of character strings
  - I think "Halpern" a very interesting pattern
  - I may want to find all occurrences of that pattern in a paper
- Other patterns:
  - **if** followed by any string of characters followed by **then**
  - all filenames ending with ".doc"

Pattern matching comes up all the time in text search.

A *finite automaton* is a particularly simple computing device that can recognize certain types of patterns, called *regular languages*

- in the next two weeks, we'll study finite automata and regular languages

Finite Automata

A *finite automaton* is a machine that is always in one of a finite number of states.

- When it gets some input, it moves from one state to another
  - If I'm in a "sad" state and someone hugs me, I move to a "happy" state
  - If I'm in a "happy" state and someone yells at me, I move to a "sad" state
- **Example:** A digital watch with "buttons" on the side for changing the time and date, or switching it to "stopwatch" mode, is an automaton
  - What's are the states and inputs of this automaton?
- A certain state is denoted the *start* state
  - That's how the automaton starts life
- Other states are denoted *final* state
  - The automaton stops when it reaches a final state
  - (A digital watch has no final state, unless we count running out of battery power.)

Representing Finite Automata Graphically

A finite automaton can be represented by a labeled directed graph.

- The nodes represent the states of the machine
- The edges are labeled by inputs, and describe how the machine transitions from one state to another

Consider the following example from Rosen (Example 4, p. 805):

.

- There are four states: s_0, s_1, s_2, s_3
  - s_0 is the start state (by convention)
  - s_0 and s_3 are the final states (denoted by double circles, by convention)
- The labeled edges represent the transitions, and describe what happens for each possible input
  - The inputs are either 0 or 1
  - For example, if the machine is
    - in state s_0 and reads 0, it stays in s_0
    - in state s_0 and reads 1, it moves to s_1
    - in state s_1 and reads 0, it moves to s_1
    - in state s_1 and reads 1, it moves to s_2

What happens on input 00000? 0101010? 010101? 11?

- Some strings move it a final state; some don't.
- The strings that take it to a final state are *accepted*.

A Parity-Checking Automaton

Here's an automaton that accepts strings of 0s and 1s that have even parity:

- An even number of 1s

We need two states:

- s_0: we've seen an even number of 1s so far
- s_1: we've seen an odd number of 1s so far

The transition function is easy:

- If you see a 0, stay where you are; the number of 1s hasn't changed
- If you see a 1, move from s_0 to s_1, and from s_1 to s_0

Here's the graph:

Finite Automata: Formal Definition

A *(deterministic) finite automaton* is a tuple $M = (S, I, f, s_0, F)$:

- S is a finite set of states;
- I is a finite input alphabet (e.g. $\{0, 1\}$, $\{a, \ldots, z\}$)
- f is a transition function; $f : S \times I \to S$
  - f describes what the next state is if the machine is in state s and sees input $i \in I$.
- $s_0 \in S$ is the initial state;
- F is the set of final states.

For the figure from Rosen:

- $S = \{s_0, s_1, s_2, s_3\}$
- $I = \{0, 1\}$
- $F = \{s_0, s_3\}$
- The transition function f is described by the graph;
  - $f(s_0, 0) = s_0$; $f(s_0, 1) = s_1$; $f(s_1, 0) = s_0$; ...

You should be able to translate back and forth between finite automata and the graphs that describe them.

Describing Languages

The *language* accepted (or *recognized*) by an automaton is the set of strings that it accepts.

- A *language* is a set of strings

We need tools for describing languages.

- If A and B are sets of strings, then AB, the *concatenation* of A and B, is the set of all strings ab such that $a \in A$ and $b \in B$.
  - **Example:** If $A = \{0, 11\}$, $B = \{111, 00\}$, then
    - $AB = \{0111, 000, 11111, 1100\}$
    - $BA = \{1110, 11111, 000, 0011\}$
- Define A^{n+1} inductively:
  - $A^0 = \{\lambda\}$: λ is the empty string
  - $A^1 = A$
  - $A^{n+1} = AA^n$
- $A^* = \cup_{n=0}^{\infty} A^n$.
  - What's $\{0, 1\}^n$? $\{0, 1\}^*$? $\{11\}^*$?