Inquiry

• What do graphs have to do with computing?
• How are packets routed on the Internet or in a network?
3. Apply the basic notions of graph theory, including by means of proof

Reading
Epp, Sec. 10.1-10.4; Sec. 12.2-12.3

Subtopic outcomes
3.1a Construct a graph from a description**
3.1b Describe a basic concept of graph theory**
3.2 Apply the concept of graph isomorphism
3.3 Describe a transition system*
3.4 Use structural induction to prove an assertion about graphs*
1. Graphs

- What does a relation *look* like?
- How do packets find their way around the Internet?
- How can we describe how to travel from one place to another?
- How do apps find a route maps?

Directed graphs (digraphs)

- Graph $G = (V, E)$: a set $V$ of vertices and a set $E$ of directed or undirected edges
- $E$ is a relation $\subseteq V \times V$
- Graphs model communication relations, state transitions, precedence in time

$V = \{a, b, c, d, e\}$

$E = \{(a, b), (b, e), (b, d), (c, d), (d, c)\}$
Undirected graphs

- Edges are segments, not arrows
- Undirected graphs are equivalent to directed ones in which all edges are double-arrowed

Applications of graphs

- Network topologies: star, linear bus, ring
- Network routing algorithms
- Scheduling (e.g., a prerequisites diagram)
- Finite automata (vertices are states; edges are transitions between states)
- Trees to represent hierarchies of all kinds
Graph-related terminology

• In this course, we use slightly different definitions from Epp’s
• A graph for us is what Epp calls a digraph
• A graph $G = (V, E)$ has a set of edges, $E \subseteq V^2$
• Hence no distinct “parallel edges” exist
• We may refer to $V(G)$, $E(G)$
• A path is a sequence of edges and we don’t refer to formally defined “walks”, “trails”, etc.
• For us a cycle is what Epp calls a “circuit”

Subgraphs

• $H = (V', E')$ is a subgraph of $G = (V, E)$ iff:
  – $H$ is a graph
  – $V' \subseteq V$
  – $E' \subseteq E$
• Example: the right graph below is a subgraph of the one on the left
Connectivity

- **Connected graph**: one that containing a path between any pair of vertices

- Not-connected example:

- **Strongly connected** graph (undirected): one in which each pair of vertices is joined by an edge

Paths

- **Path**: A sequence of pairwise adjacent vertices whose edges connect two vertices in a graph
- Path from *a* to *d*: *(a,b,d)*
Finding paths in graphs

- Standard algorithms, *breadth-first search* and *depth-first search*, visit all vertices of a graph
- *Breadth-first search* visits all vertices adjacent to starting vertex, then all vertices adjacent to these, etc.
- *Depth-first search* searches until reaching a dead end
- *Reachability matrix* of a graph, showing which vertices are reachable from which others, may be obtained from its adjacency matrix

Cycle

- *Definition*: A path from one vertex to itself that does not repeat an edge
- Cycle below: \((b, e, d)\)
- A graph with no cycles is *acyclic*
### Degree of a vertex (δ)

The number of vertices adjacent to it

- \( \delta(a) = 1 \)
- \( \delta(b) = 3 \)
- \( \delta(c) = 1 \)
- \( \delta(d) = 2 \)
- \( \delta(e) = 1 \)

**In-degree** of a directed graph’s vertex:
number of vertices it is adjacent to

**Out-degree**: number of vertices adjacent to it

---

### Degree of a graph

- The *degree of a graph* is the sum of the degrees of its vertices
- **Theorem:** \(( \forall G ) ( \delta(G) = 2 |E(G)| )\)
- **Theorem:**
  \(( \forall G )\) Even \( (|\{ x \in V(G) \text{ s.t. } \text{Odd}(\delta(x))\}|)\)
  (Any graph has an even number of vertices of odd degree)
- **Proofs:** by induction
Matrices

- An $m \times n$ matrix $A$ over set $S$ is an array of entries $a_{ij} \in S$ s.t. $i$ is a row and $j$ is a column.
- Graphs may be represented by $|V| \times |V|$ matrices whose entries denote vertices and whose entry values denote existence or weights of edges connecting vertices.
- Undirected graphs may be represented by symmetrical matrices.

Matrix representation of graphs

```java
final Graph g1 = {
   // A B C D E F G H
   {0,0,0,0,0,0,0,1},   //A
   {0,0,0,0,0,1,0},     //B
   {0,0,1,1,0,1,0},     //C
   {0,1,0,1,0,0,0,0},   //D
   {0,0,1,1,0,0,0,0},   //E
   {0,0,0,0,0,1,0},     //F
   {0,1,1,0,0,1,0,1},   //G
   {1,0,0,0,0,0,1,0}    //H
};
```
### Weighted graphs

A weighted graph has an adjacency matrix of reals or $\infty$, rather than truth values.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2.5</td>
<td>2.9</td>
</tr>
<tr>
<td>b</td>
<td>2.0</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>c</td>
<td>3.7</td>
<td>1.2</td>
<td>0</td>
</tr>
</tbody>
</table>

Weight can reflect a distance or cost.

### Directed acyclic graph (dag)

- In a dag, the edges define a *partial ordering on* the vertices.
- Vertices from which no edges extend are called *sinks*, those to which no edges point are called *sources*.
- *Example*: a listing of courses in a prerequisite relation so that no course is taken before its prerequisites.
- [pic]
Subtopic outcomes

3.1a Construct a graph from a description**
3.1b Describe a basic concept of graph theory**

2. Graph isomorphism

- Can two objects be alike if their components have different names?
- How can we tell that they are alike?
- Why does isomorphism matter?
Graph isomorphism

- “Isomorphism” is from Greek, “same form”; isomorphic objects have similar structure
- Graphs that are the same except for the names of their components are isomorphic
- Examples:

Isomorphism definition

- Graphs $G$, $G'$ are isomorphic iff there exist bijections $f$, $g$ between their sets of vertices and sets of edges, and
  $$(\forall (x, y) \in E(G)) (f(x), f(y)) = g(x, y)$$
- Theorem: Graph isomorphism is an equivalence relation
**Isomorphism invariant properties**

- If $G'$ is isomorphic to $G$, and $G$ has invariant property $P$, then $G'$ has the property.
- Some invariant properties:
  - $|V(G)| = |V(G')|$
  - $|E(G)| = |E(G')|$
  - has a vertex of degree $k$
  - has $m$ vertices of degree $k$
  - has a circuit of length $k$
  - is connected
  - has an Euler circuit (every vertex exactly once)

**Non-isomorphism examples**

Invariant properties not shared:

- $G$ has nine edges; $G'$ has eight
- $H$ has a vertex of degree four; $H'$ does not
3. Transition systems

- How can we model change, such as the transformation of data in a computation?
- What are the *simplest* imaginary computing machines like?
**Kripke structures**

- Unlabeled transition system, used to diagram and reason about reactive systems
- Kripke structure for a microwave oven:

  ![Diagram of Kripke structure for a microwave oven](image)

  **Key**
  - `start` = start-button pressed
  - `heat` = heating on
  - `shut` = door shut

**Transition systems**

- A *transition system* is a labeled digraph, where vertices denote *state* (memory), edges denote transitions and labels
- Components:
  - Alphabet $\Sigma$ (set of symbols)
  - State set (a state can be anything)
  - Transition function or relation (rules for going from one state to another)
- Applications: modeling reactive systems, stochastic processes, and kinds of computation
State-transition systems

- Let $S$ be state space of solutions to a problem, then $S_1$ are the 1-component prefixes to solutions (1-edge paths, first moves in a game, etc.)
- Let $T = \langle S, \Rightarrow \rangle$ be a transition system where
  - $S$ is a set of states
  - $\Rightarrow$ is a transition relation in $(S \times S)$
- For $q \in S$, $\text{next}(q) = \{ \sigma \in S \mid q \Rightarrow \sigma \}$
- Examples: Rules of chess, edges in graph
- A solution path is a sequence $X [1..n]$ of states that satisfies a goal constraint, and
  $(\forall i \leq n) X [i] \Rightarrow X [i + 1]$

Markov decision processes

- *Markov assumption*: the current state of a system depends only on the finite history of its previous states
- *First-order Markov process*: state depends only on previous state; second-order: state depends only on previous two states
- If approximations are too inaccurate, number of state variables or order of Markov process may be increased
Markov models

- A Markov state machine or chain is a system with a finite number of observable states; gives probability of each possible state transition
- First-order Markov chain is one where probability of present state depends only on previous state
  Example: weather at any location

Example: Weather

- Let states be \{sunny, cloudy, rainy\}
- Let transitions be as follows:

<table>
<thead>
<tr>
<th></th>
<th>sunny</th>
<th>cloudy</th>
<th>rainy</th>
</tr>
</thead>
<tbody>
<tr>
<td>sunny</td>
<td>.4</td>
<td>.5</td>
<td>.1</td>
</tr>
<tr>
<td>cloudy</td>
<td>.2</td>
<td>.5</td>
<td>.3</td>
</tr>
<tr>
<td>rainy</td>
<td>.1</td>
<td>.3</td>
<td>.6</td>
</tr>
</tbody>
</table>

- First-order
  Markov model:
Example: state of motion

- If states are running, walking, or standing still, then this MM could apply:

\[
\begin{array}{ccc}
t & R & W & S \\
t+1 & 0.5 & 0.3 & 0 \\
R & 0.5 & 0.5 & 0.75 \\
W & 0.5 & 0.5 & 0.75 \\
S & 0 & 0.2 & 0.25 \\
\end{array}
\]

- Note: columns add up to 1.0
- Graph representation of this HMM is shown above

Finite automata

- A finite automaton is a transition system with a finite set \( Q \) of states
- The edges of the graph are labeled with input symbols in alphabet \( \Sigma \); the edges define a transition function \( \delta : Q \times E \)
- If destination for each transition is unique, the automaton is called deterministic
Formal definition of DFA

• A deterministic finite automaton is a 5-tuple \( \langle Q, \Sigma, \delta, q_0, F \rangle \) where
  – \( Q \) is a finite set of states
  – \( \Sigma \) is a finite alphabet
  – \( \delta: Q \times \Sigma \rightarrow Q \) is a state-transition function
  – \( q_0 \in Q \) is the starting state
  – \( F \) is the set of accepting states

• We can also define \( \delta^* : Q \times \Sigma^* \rightarrow Q \), the reflexive transitive closure of \( \delta \), which tells what state \( \delta \) yields for a string

Some simple DFAs

(a)

(b)

(c)

(d)
3. Graphs and transition systems

FA definitions

• In Epp, accepting states are ones “when something special happens”; this may be an output in an interaction
• In the standard definition, an accepting state is one in which the finite input ends, causing acceptance of input if current state is an accepting state
• Epp’s definition of FA makes it an interactive automaton

Graph of a transition function

• A graph $G = (V, E)$ is a set of vertices and a set of ordered pairs of vertices
• $E$ is a relation on $V$
• If edges are labeled with symbols, and each edge from vertex $u$ labeled with symbol $a$ goes to a unique vertex $v$ …
• … then the labeled graph denotes a transition function $\delta : V \times \Sigma \rightarrow V$
Reflexive transitive closure

- A relation or function may be applied over and over; for example, \( f(f(f(x))) \)
- The reflexive transitive closure of a function or relation is the set of values that can be obtained by applying the function over and over in this way.
- Example: the reflexive transitive closure of the addition operation on natural numbers is the natural numbers
- \( \delta^* \) is the reflexive transitive closure of \( \delta \)

Language of a DFA

- For DFA \( M \), the language of \( M \), \( L(M) = \{ x \in \Sigma^* \mid (\delta^*(q_0, x) = q') \wedge (q' \in F) \} \)
- …where \( \Sigma^* \) is the set of all strings over \( \Sigma \), and for string \( x \), \( \delta^*(q_0, x) \) is the state reached after repeated applications of \( \delta \) to states and to elements of \( x \)
- Example: \( L(M) \) for \( M \) below is all bit strings that have a '1' followed by 0 or more '0's.
### Example DFA

- This DFA accepts strings that start with any number of 1’s (possibly none), followed by a 0, followed by any number of (10)s, followed by any number of 0s
- Some elements of its language:
  0, 10, 00, 100, 110, 010, 1010, 10100

### Example regular expressions
(What are their DFAs?):
- \((01)^*\) : all strings that repeat the string 01, zero or more times
- \(01^* \mid 10^*\) : all strings that either consist of a 0 followed by zero or more 1’s, or consist of a 1 followed by zero or more 0’s
Stack machines (pushdown automata)

- PDA $A = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$
- As with DFAs, $Q$, $\Sigma$, $q_0$, $F$ are state set, alphabet, start state, accept state set, resp.
- $\Gamma$ (Gamma) is a set of stack symbols
- $\delta: (Q \times \Sigma \times \Gamma \cup \{\lambda\}) \times (Q \times \Gamma \cup \{\lambda\})$ is the transition relation
- PDA reads a $\Sigma$ symbol, popping a $\Gamma$ symbol from stack, writes a $\Sigma$ symbol, pushes a $\Gamma$ symbol
- Acceptance of an input string requires empty stack on termination of input

Turing machines

- Based on human “computer” with paper and pencil
- Operations: tape head reads a symbol at current location on paper, moves left or right, writes symbol at current location
- Next action is looked up in transition table based on current input and “state of mind” of computer
- Machine halts when a it enters a “halting” (accept or reject) state
3. Graphs and transition systems

**TM definition**

- **Turing machine**: \( M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} \rangle \)
  where \( \Sigma \) and \( \Gamma \) are input and tape alphabets

- **Transition function** \( \delta \):
  \[
  \begin{align*}
  Q \times (\Sigma \cup \{\#\} \cup \Gamma) & \rightarrow Q \\
  Q \times (\Sigma \cup \Gamma \cup \{\#\}, L, R) & \rightarrow Q
  \end{align*}
  \]
  where \( L, R \) denote left or right moves

- Tape is infinite in both directions
- Tape head starts at leftmost nonblank cell
- First blank to right of a symbol in \( \Sigma \) has infinitely many blanks to its right

**Example: negater TM**

- The TM above reads one bit, writes its negation to the tape, and halts
- \( \Sigma = \{ 1, 1 \} \)
- \( Q = \{ q_0, q_1 \} \)
- \( \delta = \{ \langle(q_0, 0), (q_1, 1) \rangle, \langle(q_0, 1), (q_1, 0) \rangle \} \)
Finite transducers

- A finite-state transducer may have output symbols as part of its labels
- FTs are called *Mealy* and *Moore* machines
- On a given input, an FST will output a string of the same length
- Hence it is not an *accepter*, but performs *transduction* from one string to another

Subtopic outcome

3.3 Describe a transition system*
4. Structural induction

• How is a computing system verified?
• How can properties of languages and transition systems be proven?

Proving correctness of DFAs

• We use structural induction on the size of input strings to show that a given DFA accepts precisely a certain language

• General form of the assertion to be proven:

\[(\forall x \text{ s.t. } |x| = n) \ x \in L \iff x \in L(M) \Rightarrow (\forall x \text{ s.t. } |x| = n + 1) \ x \in L \iff x \in L(M)\]
Example proof of correctness

- *M* (below) accepts $L = 10^*$

```
  a  1  b
  \  \   \  \ 0
   \   \```

- **Base case:** ‘1’ $\in L(M)$
- **Induction:** From accept state b, adding a 0 keeps string in $L$, whereas adding a 1 triggers rejection

Structural induction

- Many sets, including strings, trees, and formal languages, may be defined recursively
- Method for showing that recursively defined set $S$ has property $P$:
  1. Show that $P(x)$ for each element of the base case of $S$
  2. Show that for each recursive rule, applying the rule to an element that satisfies $P$ yields an object that also satisfies $P$
Example proof of correctness

- Language specification: \( n_1(x) = 1 \)
- Regular expression: \( 0^*10^* \)
- DFA:

```plaintext
Diagram of DFA with transitions:
0 -> 1
1 -> 0
```

- Proof of correctness:
  - **Base**: \( \lambda \not\in L(A) \), because left state rejects
    \( 1 \in L(A) \), because middle state accepts
  - **Induction**: 1 takes accepted string to reject state; 0 takes any string to same state

---

Structural induction on string length

- Let language \( L = \{\lambda\} \cup \{x \mid x \in L\} \cup \{yz \mid y, z \in L\} \)
- **Theorem**: Every element of \( L \) has equal numbers of left and right parentheses (property \( P \))
- **Proof**:  
  - **Base**: \( \lambda \) has the property \( P \)
  - **Induction**:  
    1. \( P(x) \implies P((x)) \) adding one left, one right yields balance  
    2. \( P(x) \land P(y) \implies P(xy) \) concatenating two strings, each with equal left and right, yields balance
3.4 Use structural induction to prove an assertion about expressions or graphs*

References


