Lecture 1: Mathematical Review

Michael Engling

Department of Computer Science
Stevens Institute of Technology

CS 334 Automata and Computation
Fall 2015

OUTLINE: Review of some CS135 Stuff

Sets–Collect Them All!

Sequences and Tuples–Putting Things in Order

Functions and Relations–Maps and More

Graphs – (Sets of) Vertices and Edges

Strings and Languages–Sequences of Symbols, Sets of Strings

Boolean Logic–Well, Yes and No

Terminology and Nomenclature

Sets–Collect Them All!

A set is an unordered collection of objects called *elements*.

We name them, typically, with capital Latin letters:

$$A = \{1, 2, 3, 4\}, B = \{3, 2, 1\}$$

A set is a subset of another set if every element of the first is also an element of the second. Here, $B \subseteq A$.

A proper subset is a subset which is not equal to the parent set. Here we have $B \subset A$, also.

The empty set, $\{\} = \emptyset$, has no elements. It is a subset of every set.

The cardinality of a (finite) set is the number of elements it contains: $|A| = 4, |B| = 3, |\emptyset| = 0$.

We will sometimes refer to sets with just one element as a *singleton*.

We will be interested in sets with infinitely many elements. We will need to take care to describe these with some precision.

Operations on Sets

We'll concern ourselves with several basic operations on sets:

Intersection: $A \cap B$ consists of all elements in BOTH A and B.

Union: $A \cup B$ consists of all elements in either A OR B (or both).

The complement of a set A, written $\overline{A}$ consists of elements which are *not* elements of A. We will have much reason to use this idea, but it is dependent on the notion of a universal set U which consists of all the relevant elements in a given context.

We'll have a couple of "new" operations, but we're getting ahead of ourselves.

You have likely seen Venn Diagrams at some point. We might make use of them casually, informally. Sipser does, at least a little.

Sequences and Tuples–Putting Things in Order

The notion of a sequence implies order. With sets $\{1, 2, 3, 4\} = \{4, 3, 2, 1\}$, but the 4-tuples $(1, 2, 3, 4) \neq (4, 3, 2, 1)$.

We will deal with k-tuples, $k \in N$. A 2-tuple, (n, m) is an ordered pair.

We will often create tuples from Cartesian products (cross products).

The Cartesian product, $A \times B$ consists of all ordered pairs in which the first element comes from A and the second from B.

So $\{1, 2, 3\} \times \{n, m\} = \{(1, n), (1, m), (2, n), (2, m), (3, n), (3, m)\}$.

$|A \times B| = |A| \times |B|$.
The notion of Cartesian product extends to higher dimensions as well.

Functions and Relations–Maps and More

A function from one set to another (or back to itself) is a *mapping* that assigns to each element of the *domain* a unique element of the *range*, also called the *co-domain*.

We will often express the notion like this:

$$D : Z \to Z, D(n) = 2n.$$

Here D maps the integers to themselves by doubling. An easy function to grasp.
A function is said to be *1-1* if no 2 elements of the domain are mapped to the same element of the range. (Having landed somewhere, you can tell for certain whence you started.)

A function is called *onto* if every element of the range is mapped to from some element of the domain. (Every target element gets hit.)

Transition Functions

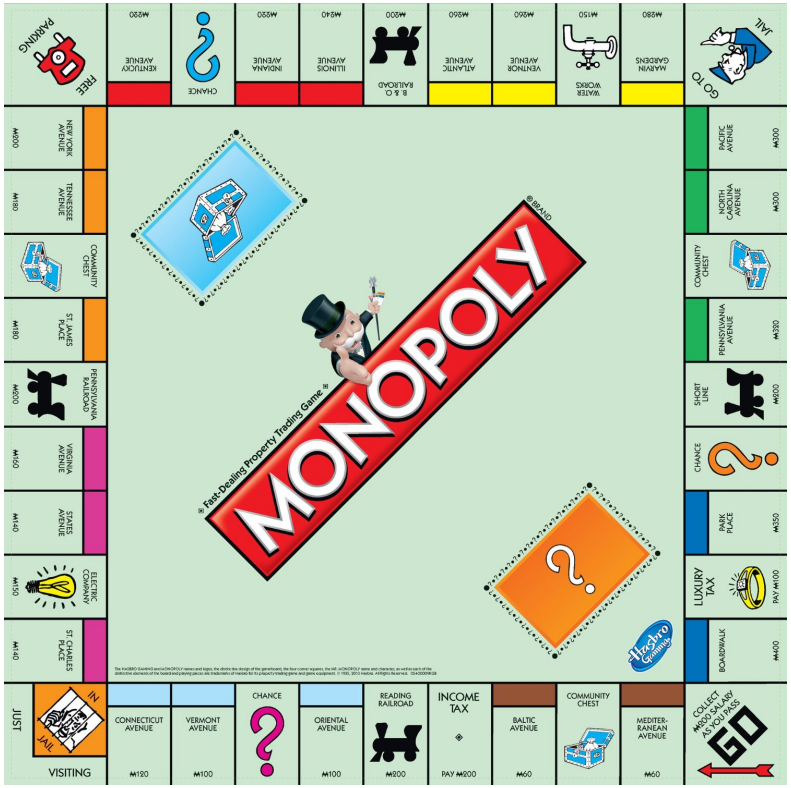
We will make great use of *transition* functions.
These will map Cartesian products to one of the sets in the Cartesian product, like this:

$$T : A \times B \rightarrow A$$

A useful example might be A = category of rental car and B = member of our Frequent Renters Program. The transition function might be called U for "Upgrade":

$$\begin{aligned} U &: (\text{economy, yes}) \rightarrow \text{midsize} \\ U &: (\text{economy, no}) \rightarrow \text{economy} \\ U &: (\text{midsize, yes}) \rightarrow \text{luxury} \\ U &: (\text{midsize, no}) \rightarrow \text{midsize} \\ U &: (\text{luxury, yes}) \rightarrow \text{luxury} \\ U &: (\text{luxury, no}) \rightarrow \text{luxury} \end{aligned}$$

MONOPOLY: $T : P \times R \to P$



$$
\begin{array}{ccc}
(\text{Go},2) & \to & \text{CommChest} \\
(\text{Go},3) & \to & \text{Baltic Ave.} \\
\vdots & \vdots & \vdots \\
(\text{Go},12) & \to & \text{ElectriCo} \\
\vdots & \vdots & \vdots \\
(\text{Jail},10) & \to & \text{FreeParking} \\
\vdots & \vdots & \vdots \\
(\text{FreeParking},8) & \to & \text{H}_2\text{OCo} \\
\vdots & \vdots & \vdots \\
(\text{BWalk},12) & \to & \text{St.Chuck Pl.}
\end{array}
$$

Figure: Monopoly Board and (part of) Transition Function

Graphs – (Sets of) Vertices and Edges

We'll make use of some graph theory terminology, even if we abuse it a bit from time to time.

Formally, a graph is two sets; the first is V, the set of *vertices* (or *nodes*); the second is E, the set of *edges* which consists of pairs of vertices. So, $G = (V, E)$.

In a *directed* graph or *digraph*, these pairs are ordered. We depict such (directed) edges with arrows.

We will permit edges in which both vertices are the same; we will call these edges *self-loops* or just *loops*.

We will almost always label the edges of our graphs. Such a labeled graph is called a *labeled* graph. (It is also sometimes called a *network*.)

Graphs – Dots and Lines

A *path* is a sequence of vertices in which each consecutive pair forms an edge.

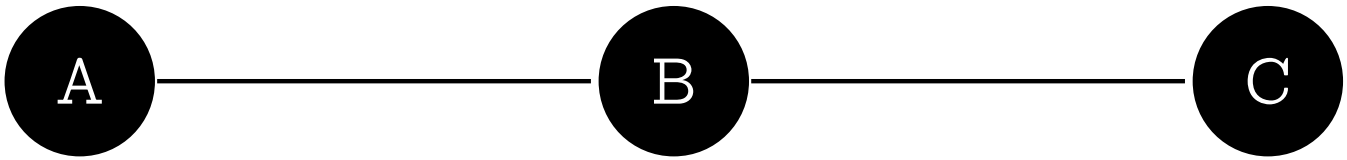
A *simple path* repeats no edges.

A *cycle* is a path beginning and ending with the same vertex.

A *tree* is a graph with no cycles, an *acyclic* graph.

The *degree* of a vertex is the number of edges in which it is an element. In a directed graph, we may distinguish between indegree and *outdegree*.

Graphs – Dots and Lines



Here $V = \{A, B, C\}$ and $E = \{\{A, B\}, \{B, C\}\}$

Strings and Languages–Sequences of Symbols, Sets of Strings

We will focus heavily on strings and languages.

A *string* is a finite sequence of symbols from a (non-empty) set called an *alphabet*, usually identified by capital Greek letters $\Sigma, \Gamma,$ etc.

As examples, 0011 and 1100 are strings over the binary alphabet, $\Sigma = \{0, 1\}$. The word "sequence" in the defintion implies order, and hence $0011 \neq 1100$.

A *language* is a set of strings (over some alphabet).

A string z is a *substring* of string w if z appears consecutively in w. Hence, 101 is a substring of 10101, but 00 is not.

Strings and Languages

The *length* of a string is the number of symbols in the string (including repetitions). The empty string, which we will depict as ε, has length zero. (Other sources may use *lambda* as the empty string.)

We will sometimes *concatenate* strings to make (potentially) longer strings. If $w = 0011$ and $z = 1100$, then:

$$wz = 00111100$$
$$zw = 11000011$$
$$z\varepsilon w = zw = 11000011$$
$$zzz = 110011001100$$

In ordering strings it will be usful for us to modify *lexicographical* (dictionary) order to *string* (*shortlex*) order which prioritizes length over content, thus:
Over $\Sigma = \{0, 1\}$ we get $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots$.

Boolean Logic–Well, Yes and No

Boolean means two-valued.
We may use **true/false**; Sipser uses 0/1.

The **negation** operation changes or flips a Boolean value. So, $\neg 0 = 1$ and $\neg 1 = 0$.

The **AND** operation $P \wedge Q$ results in **true** iff both operands P and Q are **true** while the **OR** operation $P \vee Q$ results in **true** iff either (or both) is **true**.

The implication $P \rightarrow Q$, read "if P then Q" is *true* unless P is *true* and Q is *false*.

The bi-implication, $P \leftrightarrow Q$, read "P if and only if Q", is *true* when both operands have the same value. It is the negation of the exclusive or, or **XOR**.

Boolean Logic–Well, Yes and No

DeMorgan's Laws:

$$\neg(A \vee B) = \neg A \wedge \neg B$$
$$\neg(A \wedge B) = \neg A \vee \neg B$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

Distributive laws:

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$
$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

Terminology and Nomenclature

Page 16 of Sipser has 42 definitions we may or may not need. Think of it as containing the answers to "Life, the Universe, and Everything."

Putting It All Together

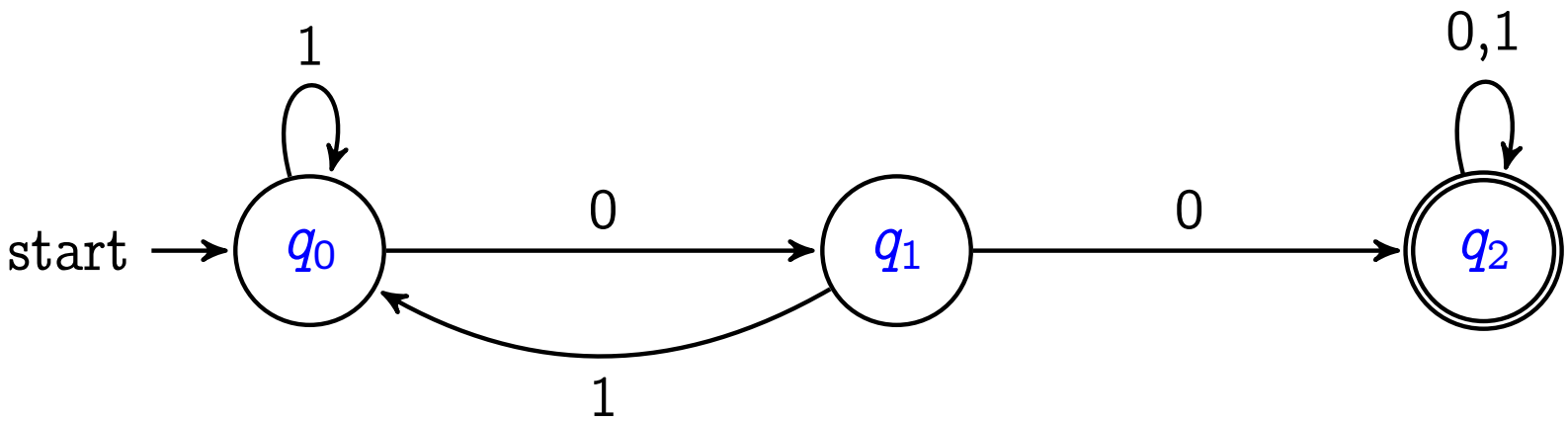
We will formally define our first automaton like this:

$$A = (Q, \Sigma, \delta, q_0, F)$$

This is a 5-tuple in which:
Q is a finite set (of states)
Σ is an alphabet ((finite) set of symbols)
δ is a transition function, $\delta : Q \times \Sigma \to Q$ (Think: Monopoly)
$q_0 \in Q$ (our initial state), and
$F \subseteq Q$ (our "good" final states).

Our First Automaton

$$A = (Q, \Sigma, \delta, q_0, F)$$



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

$\delta : Q \times \Sigma \to Q$

$q_0 = q_0$

$F = \{q_2\}$

$\delta((q_0, 0)) \to q_1 \quad \delta((q_0, 1)) \to q_0$

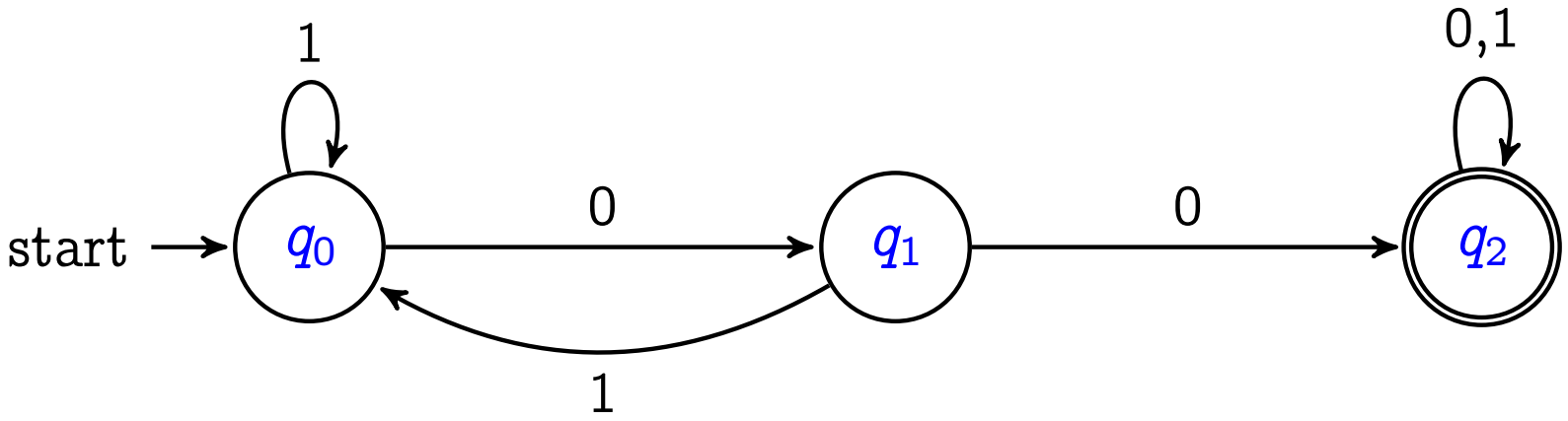
$\delta((q_1, 0)) \to q_2 \quad \delta((q_1, 1)) \to q_0$

$\delta((q_2, 0)) \to q_2 \quad \delta((q_0, 1)) \to q_2$

See why q_2 is special?

Our First Automaton

$$A = (Q, \Sigma, \delta, q_0, F)$$



$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

δ :

	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_2

$q_0 = q_0$

$F = \{q_2\}$