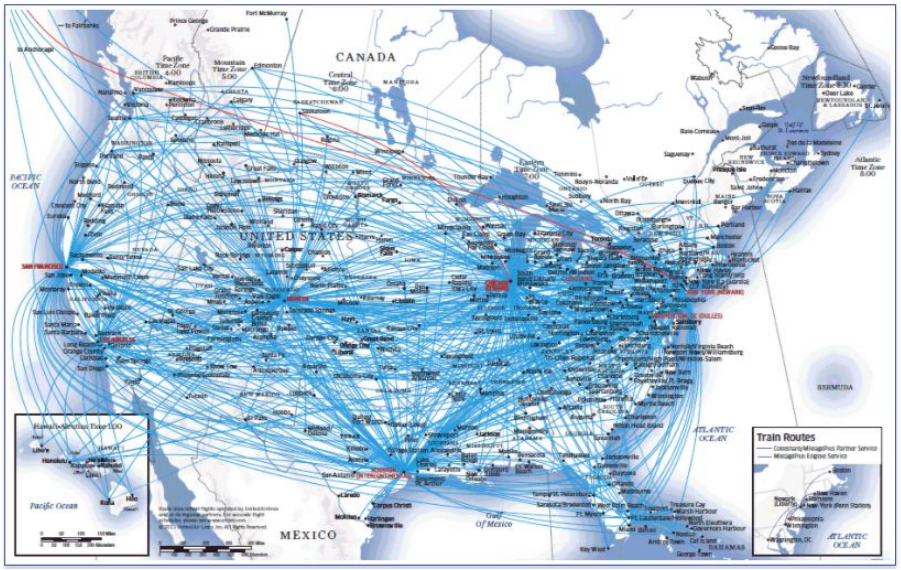
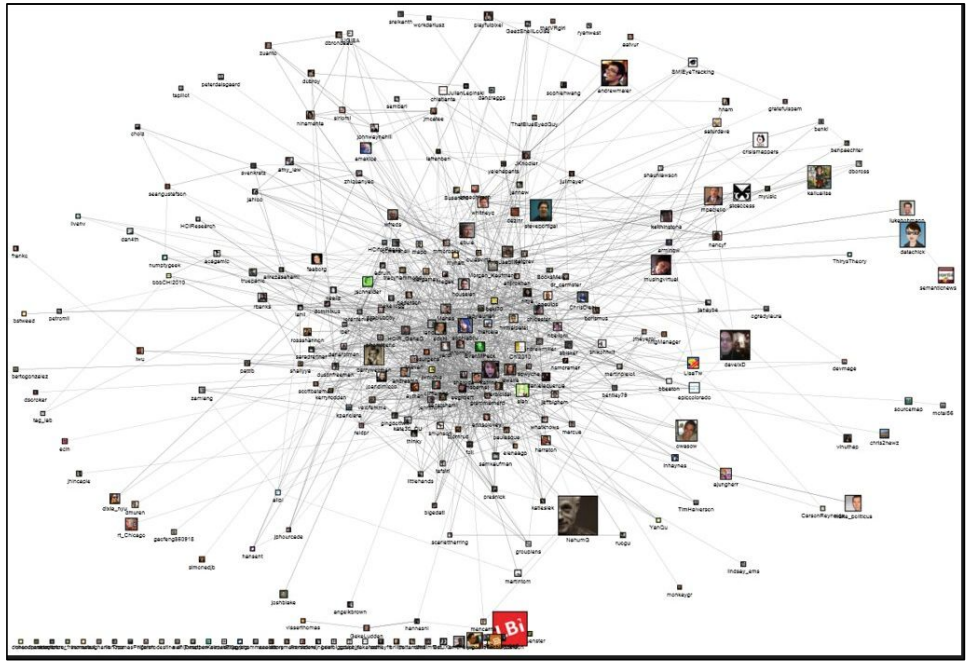
Large Graphs:
Representing Sets of States



United Airlines USA route map

Zurich downtown transit



Twitter follower connections among people who tagged the same conference

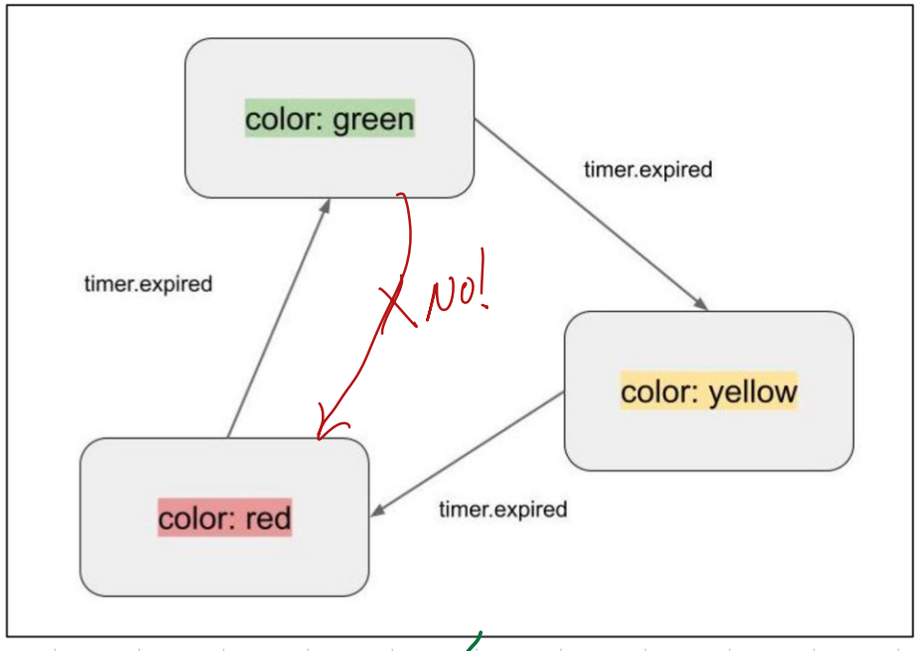
GRAPHS SO BIG THAT TECHNIQUES WE KNOW FOR STORING/TRAVERSING DON'T ALWAYS WORK!

Source URLs in slide notes

URL from today's lecture

- Stanford Large Network Dataset Collection
  - http://snap.stanford.edu/data/index.html

USING GRAPHS TO MODEL SYSTEMS + SOFTWARE



State machine: use graph to represent a system or process

Vertex: state system is in, properties about the system (color of light)

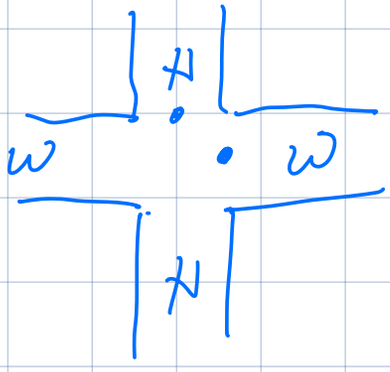
Edge: events/transitions between states, represents the sequence of steps the system may take

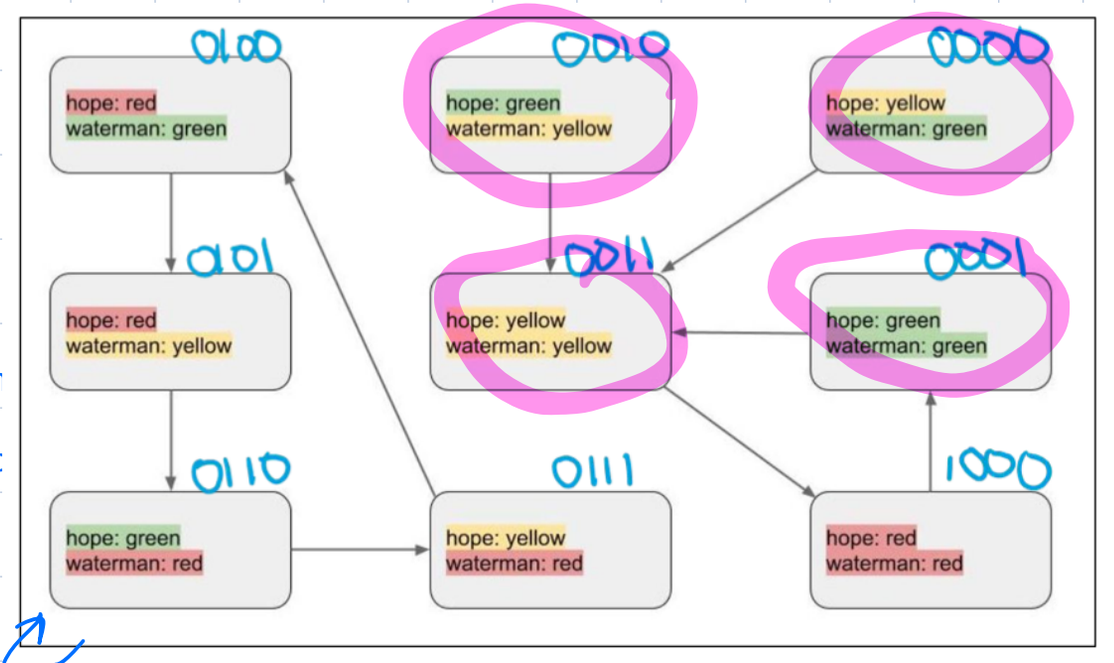
=> Important for embedded systems, testing, and critical applications—can use tools to check that system behaves according to certain constraints

↳ 3 POSSIBLE STATES: RED, GREEN, YELLOW

WHAT ABOUT TWO LIGHTS?

NOW: 3*3 = 9 POSSIBLE STATES!





Discuss: What would it mean for this syster to be unsafe?
=> Don't want two lights to be green or yellc at the same time.
(At least one light must be red at all times)

How would you compute the reachable states from a starting state?
What are some graph algorithms we remember?
- BFS
- DFS
- Dijkstra - weighted graphs
- MST - trees

BFS ends up being the better choice

We can say a system is unsafe if we could reach an unsafe state from any starting state
==> This graph is okay so long as we start in one of the safe states

So how do we do this for representing lots of states?
List[node], set[node], …

Q from lecture: If the system should never be in an unsafe state, why are those states on the graph at all? Why not just have a graph of the 4 safe states?

```
Light_State: {red, green, yellow}

# Two variables
Light_state Hope = Red
Light_state Waterman = Yellow

if timer_expires
    Hope = red

# . . .

if something_else
    Hope = red
    Waterman = yellow

# . . .

if yet_something_else
    Waterman = green
```

Imagine we had code to control these two traffic lights, perhaps with various statements throughout the code that changed them.

Since we could write any program we want, each variable could take any combination of {red, yellow, green}, so any of the 9 combinations are technically possible! Even if we don't want this behavior to happen, it's possible a bug could place the program in an unsafe state!

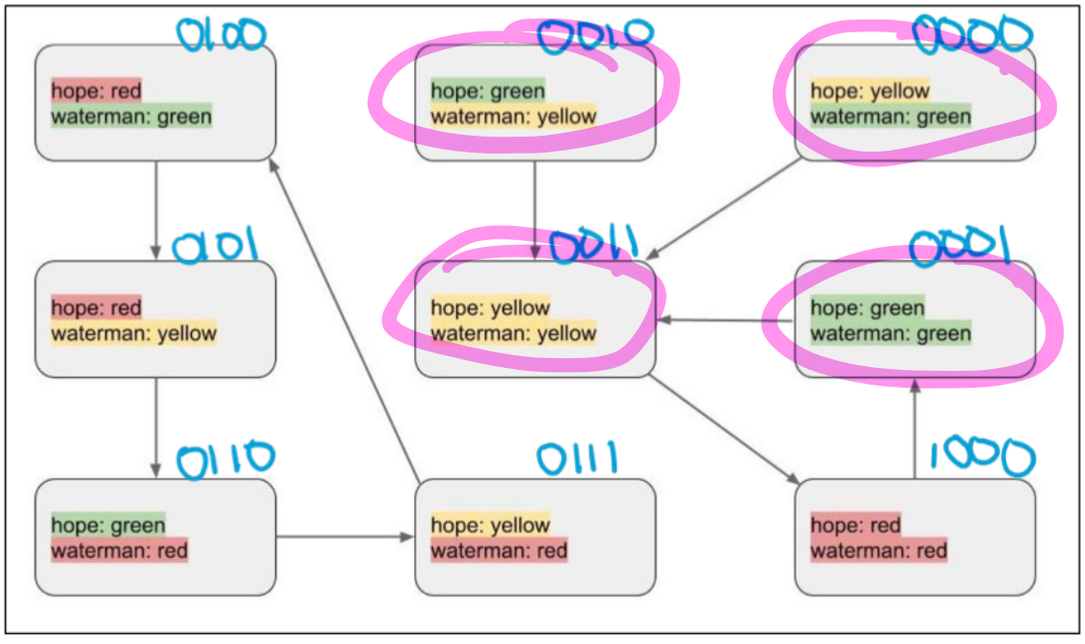
This is where modeling programs can help: if we can express the program's possible states as a state machine, we can use what we know about graphs (determining reachability) to check that the system obeys certain properties!

How do we store the set of reachable states?

Important for formal methods: representing set of states succinctly…can have a lot of states!

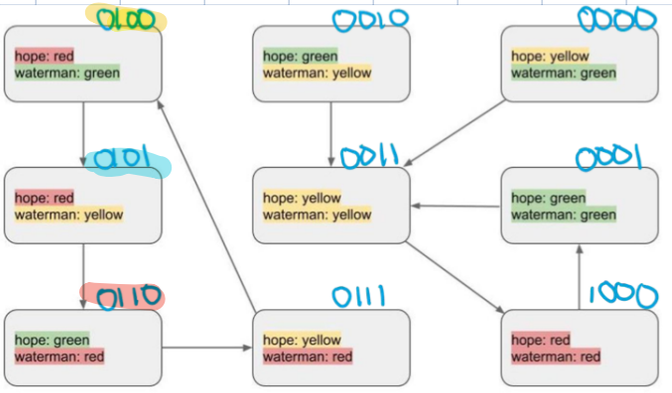
List[state], set[state]…
=> Linear in terms of “space complexity”: how much memory is required

2^200 atoms in universe
CPU: brain of computer: 2^2M states



Need: succinct way to represent states

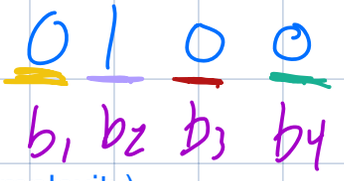
Note 4/17: We'll continue from here on Wednesday (4/19)



Idea: Give each state a name
- Smallest piece of information in a computer is a "Bit" (0 or 1)
- Assign name to each state is a "bit string", or a sequence of bits

For N states, how many bits would I need?
ceil(log2(n))

=> Represent each state as a sequence of 0's and 1's



What can we do with this?
- Represent states VERY succinctly (lower space complexity)
- Can represent set of reachable states as a decision tree => Binary Decision Diagram

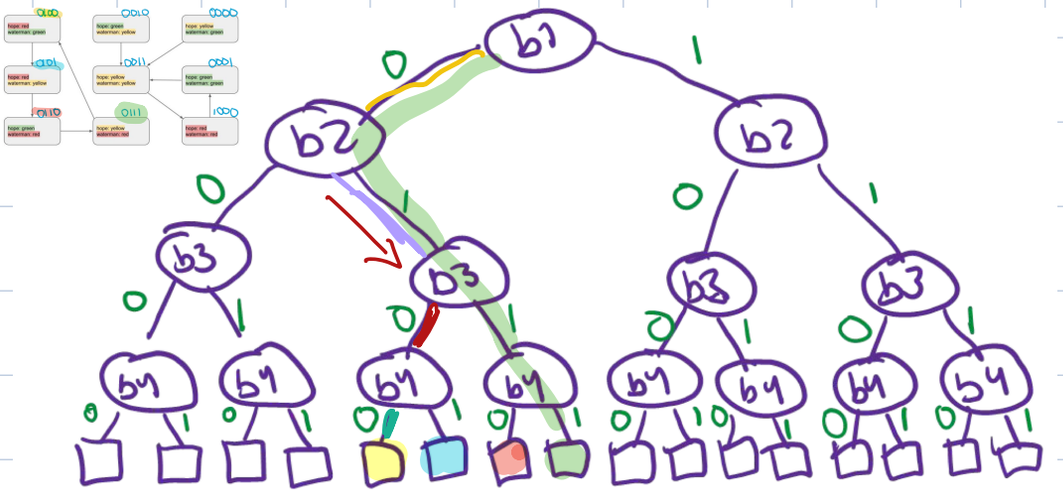
---

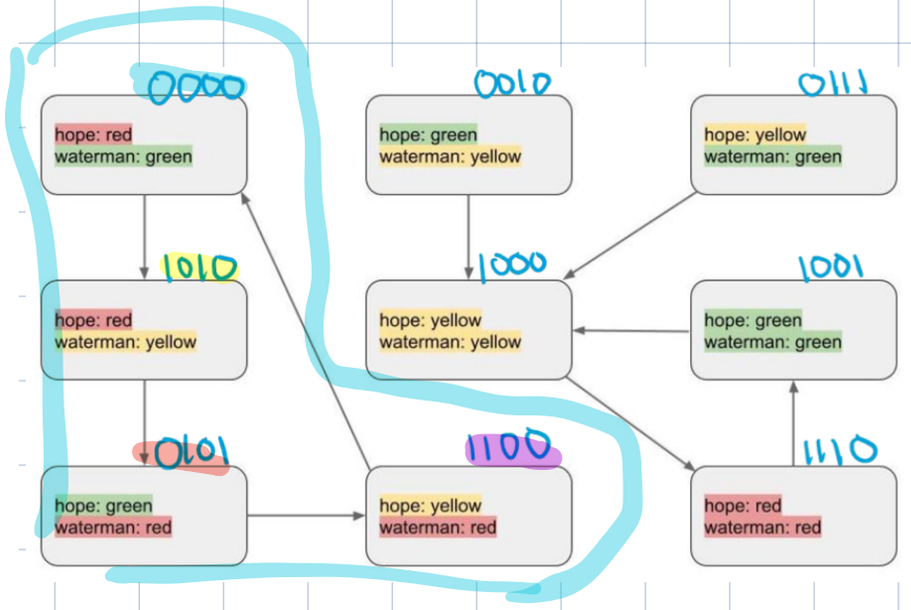
BDD: How it works (high level)
- Run BFS to find reachable states
- Mark each leaf of BDD as reachable for each state
- Can make a logical formula (a boolean expression) to represent reachable states

Reachable: (b1 == 0) AND (b2 == 1)

Could also write as: (not b1) and (b2)

==> To check if the state is safe, Don't even need to store a list of reachable states! Just need to check if the bitstring matches the formula!





Try it: what would the BDD look like if we picked these names instead?

B_1 B_2 B_3 B_4

Formula would be a lot more complicated!
((not b1) and (not b2) and (not b3) and (not b4)) OR
((b1) and (not b2) and (b3) and (not b4) OR
…

<u>Takeaways:</u>

- Coming up with a good naming of states is nontrivial
  => Problems are provably computationally hard, people work on this for specific settings

- Depending on what the BDD looks like, logical formula may get complex

Want to learn more?
Consider taking: Logic for Systems
=> Learn about algorithms people have made to represent these in more concise ways, make tools to help, software to use