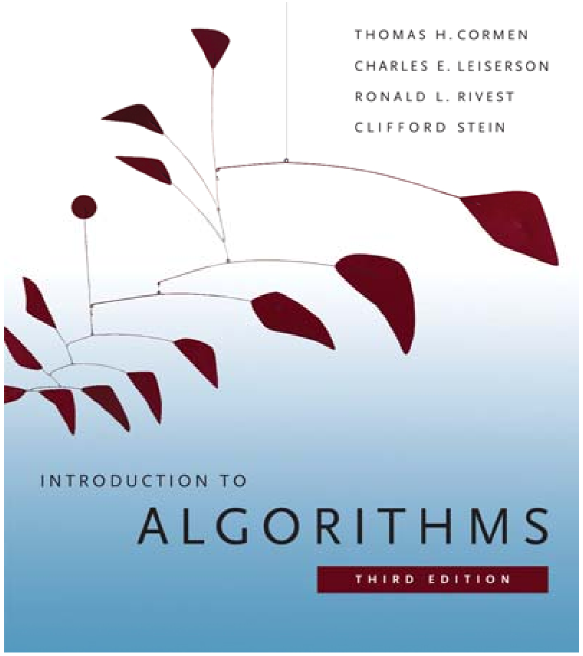
# 6.006- *Introduction to Algorithms*




## *Lecture 13*

**Prof. Manolis Kellis**

**CLRS 22.4-22.5**

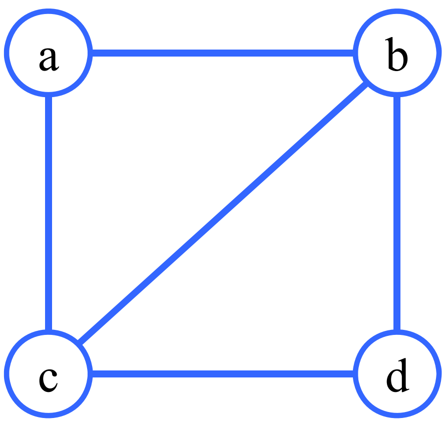
# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS). DFS vs. BFS***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Graphs

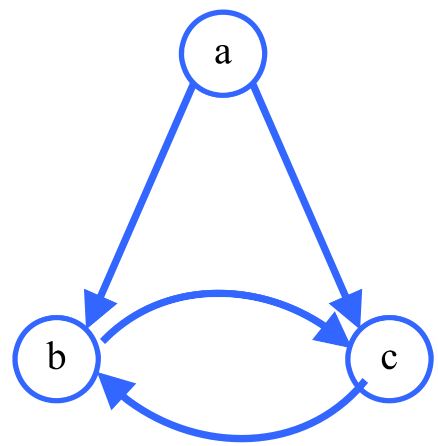
- G=(V,E)
- V a set of vertices
  - Usually number denoted by n
- E $\subseteq$ V ´ V a set of edges (pairs of vertices)
  - Usually number denoted by m
  - Note m $\leq$ n(n-1) = O($n^2$)

**Undirected example**



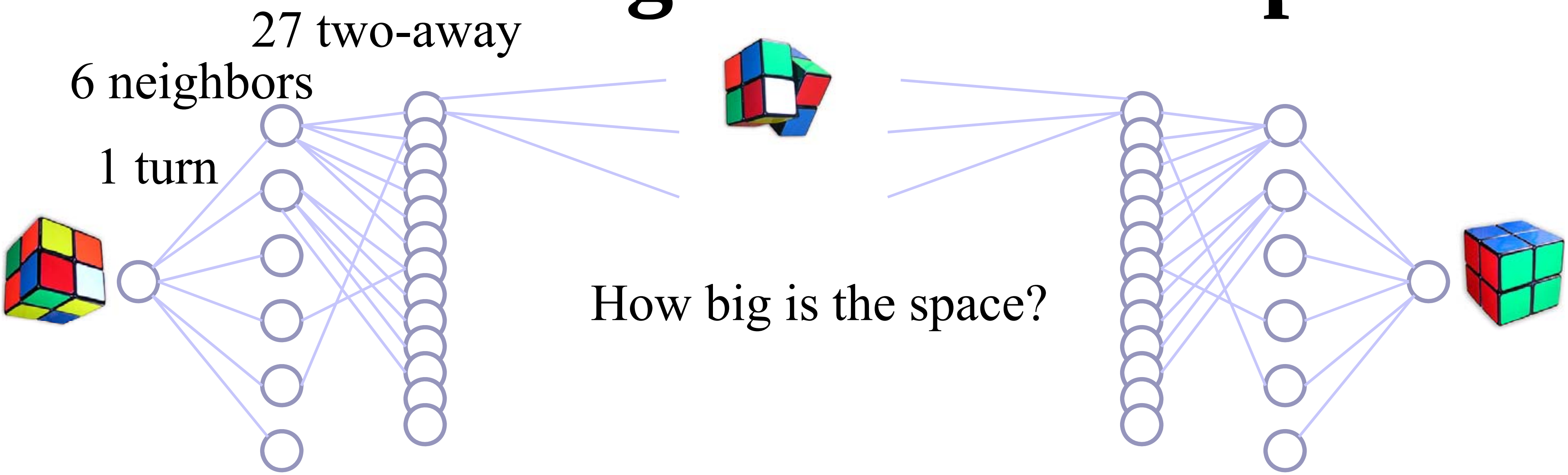

- V={a,b,c,d}
- E={{a,b}, {a,c}, {b,c}, {b,d}, {c,d}}

**Directed example**




- V = {a,b,c}
- E = {(a,c), (a,b) (b,c), (c,b)}

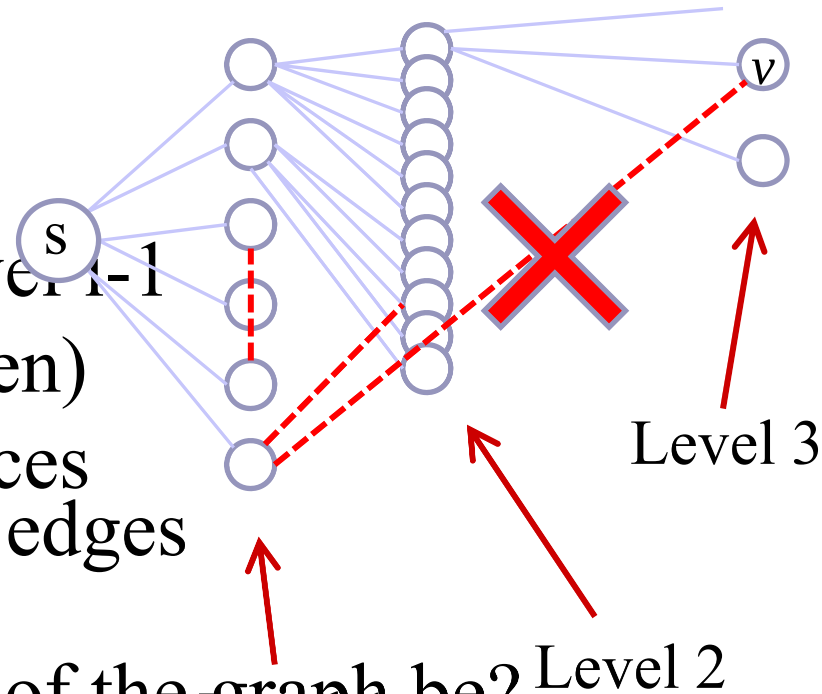
# Searching for a solution path




- Graph algorithms allow us explore space
  - Nodes: configurations
  - Edges: moves between them
  - Paths to ‘solved’ configuration: solutions

# BFS algorithm outline

- Initial vertex s
  – Level 0
- For i=1,…
  grow level i
  – Find all neighbors of level i-1
  – (except those already seen)
  – i.e. level i contains vertices reachable via a path of i edges and no fewer




- Where can the other edges of the graph be?
  – They cannot jump a layer (otherwise *v* would be in Level 2)
  – But they can be between nodes in same or adjacent levels

# BFS Algorithm

- BFS(V,Adj,s)

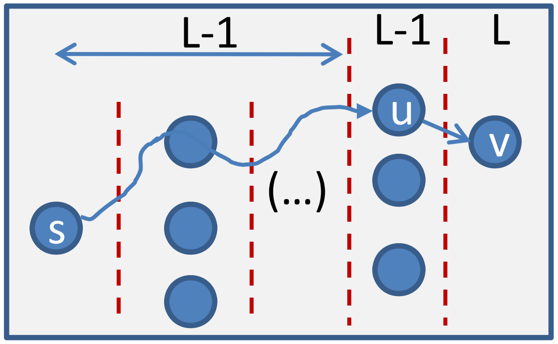
```
level={s: 0}; parent = {s: None}; i=1
frontier=[s]                           #previous level, i-1
while frontier
    next=[]                            #next level, i
    for u in frontier
        for v in Adj[u]
            if v not in level          #not yet seen
                level[v] = i           #level of u+1
                parent[v] = u
                next.append(v)
    frontier = next
    i += 1
```

# BFS Analysis: Correctness

i.e. why are all nodes reachable from s explored?

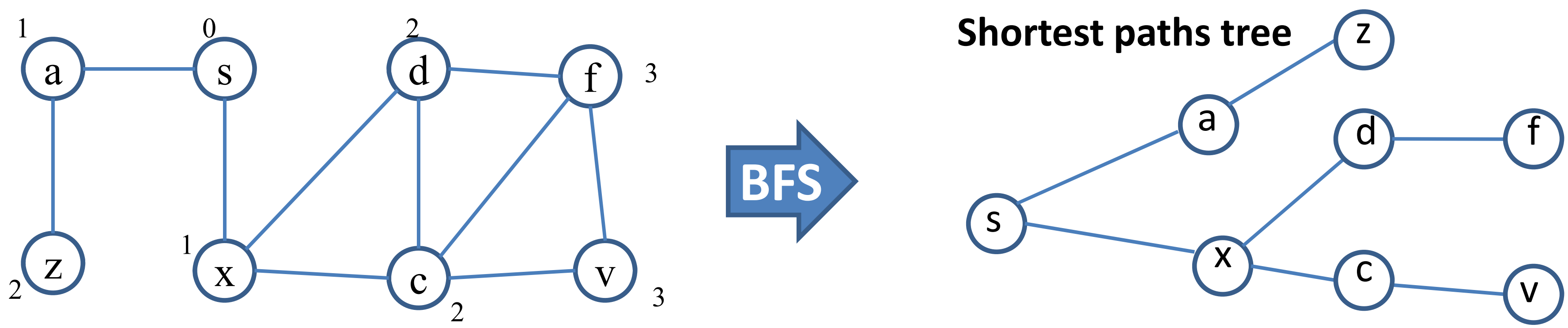
(we'll actually prove a stronger claim)

- **Claim**: If there is a path of L edges from s to v, then v is added to *next* when i=L or before
- **Proof**: induction
  - **Base case:** s is added before setting i=1
  - **Inductive step when i=L:**
    - Consider path of length L from s to v
    - This must contain: (1) a path of length L-1 from s to u
    - (2) and an edge (u,v) from u to v
  - By inductive hypothesis, u was added to *next* when i=L-1 or before
    - If v has not already been inserted in *next* before i=L, then it gets added during the scan of Adj[u] at i=L
  - So it happens when i=L or before. QED



# Corrollary: BFS→Shortest Paths

- From correctness analysis, conclude more:
  - Level[v] is length of shortest s→v path
- Parent pointers form a shortest paths tree
  - i.e. the union of shortest paths to all vertices
- To find shortest path from s to v
  - Follow parent pointers from v backwards
  - Will end up at s



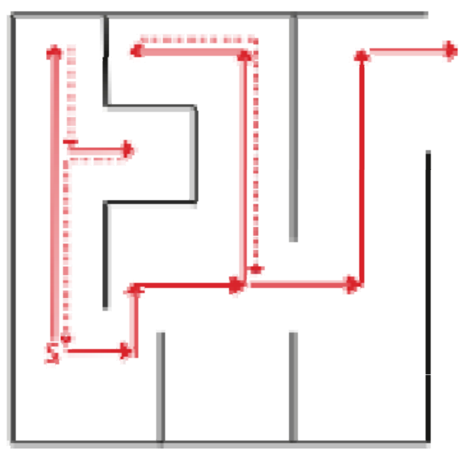
# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  – Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS). DFS vs. BFS***
  – Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  – Topological Sort on DAGs, job scheduling
  – Connected components, strongly connected
- ***Properties of real-world & biological networks***
  – Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Depth First Search (DFS)

# DFS Algorithm Outline

- Explore a maze
  - Follow path until you get stuck
  - Backtrack along breadcrumbs till find new exit
  - i.e. recursively explore

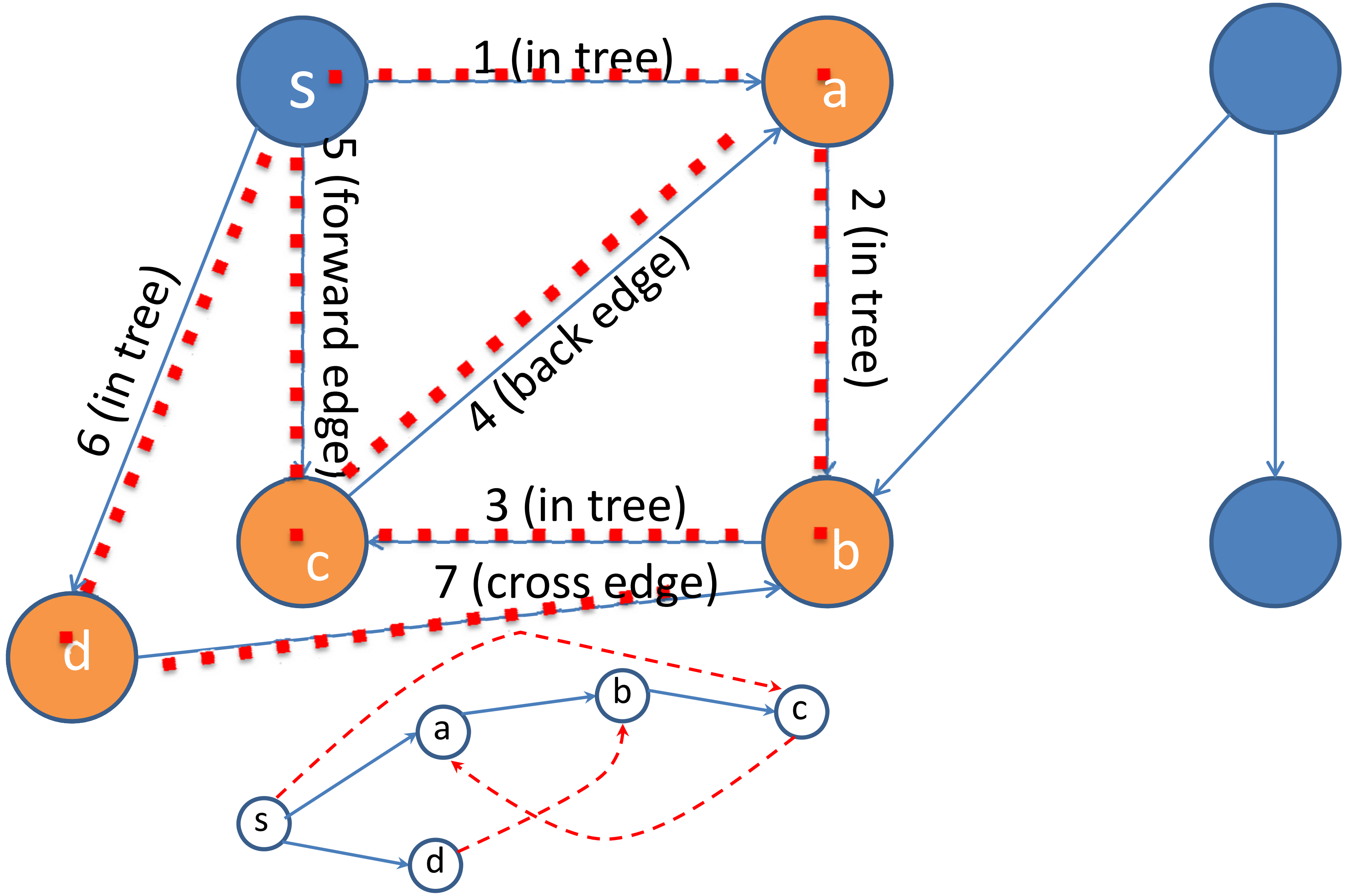


# DFS Algorithm

- ***parent*** = {s: None}
- call ***DFS-visit*** (V, Adj, s)

```
def DFS-visit (V, Adj, u)
    for v in Adj[u]
        if v not in parent                  #not yet seen
            parent[v] = u
            DFS-visit (V, Adj, v)       #recurse!
```

# DFS example run (starting from s)



# DFS Runtime Analysis

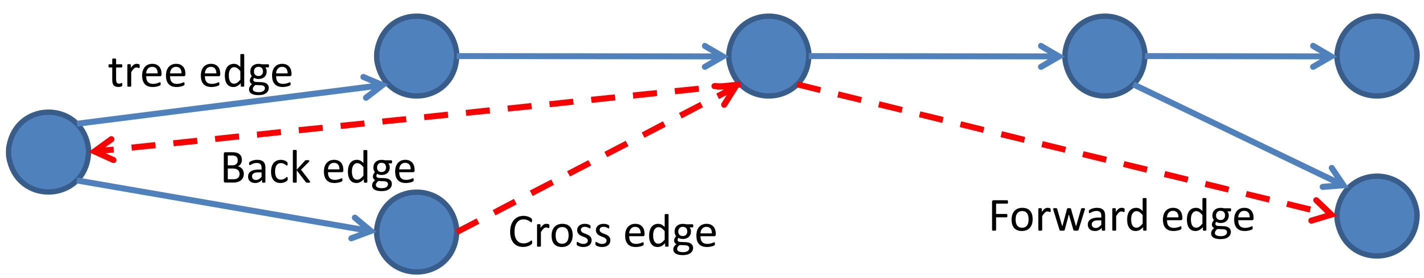
- Quite similar to BFS
- DFS-visit only called once per vertex v
  - Since next time v is in ***parent*** set
- Edge list of v scanned only once (in that call)
- So time in DFS-visit is:
  - 1 per vertex + 1 per edge
- So time is O(n+m)

# DFS Correctness?

- Trickier than BFS
- Can use induction on length of ***shortest*** path from starting vertex
  - Inductive Hypothesis:
    "each vertex at distance k is visited (eventually)"
  - Induction Step:
    - Suppose vertex v at distance k.
      - Then some u at ***shortest*** distance k-1 with edge (u,v)
      - Can decompose into s→u at ***shortest*** distance k-1, and (u,v)
    - By inductive hypothesis: u is visited (eventually)
    - By algorithm: every edge out of u is checked
      - If v wasn't previously visited, it gets visited from u (eventually)

# Edge Classification

- Tree edge used to get to new child
- Back edge leads from node to ancestor in tree
- Forward edge leads to descendant in tree
- Cross edge leads to a different subtree
- To label what edge is of what type, keep global time counter and store interval during which vertex is on recursion stack

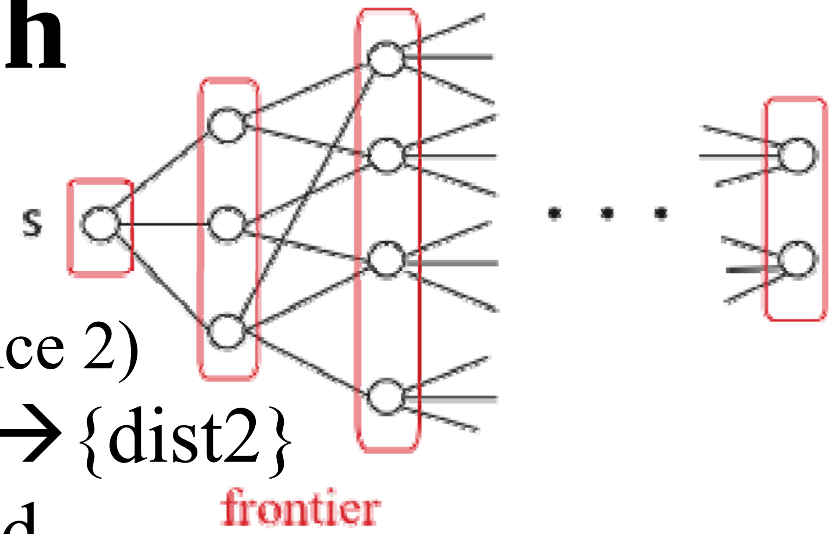


# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS). DFS vs. BFS***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

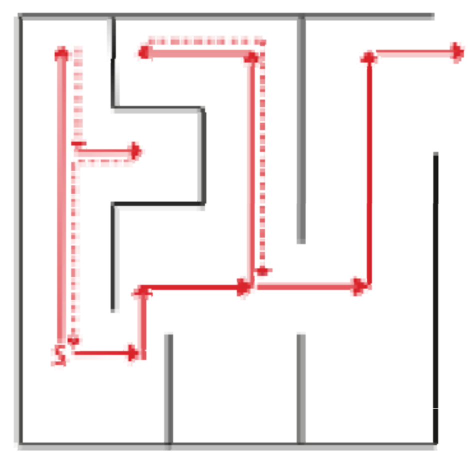
# BFS vs. DFS

# Breadth First Search




- start with vertex v
  - list all its neighbors (dist 1)
  - then all their neighbors (distance 2)
- Define frontier {s}→{dist1}→{dist2}
- Repeat until all vertices found

# Depth First Search




- Like exploring a maze
- From current vertex, move to another
- Until you get stuck
- Then backtrack till new place to explore

# BFS/DFS Algorithm Similarities

- Maintain “todo list” of vertices to be scanned

---

- Until list is empty
  - Take a vertex v from front of list
  - Mark it scanned
  - Examine all outgoing edges (v,u)
  - If u not marked, add to the todo list
    - BFS: add to end of todo list (***queue*: FIFO**)
    - DFS: add to front of todo list (***recursion stack*: LIFO**)

# Key difference: Queue vs. Stack

- BFS queue is explicit
  - Created in pieces
  - (level 0 vertices) . (level 1 vertices) . (level 2 vert…
  - the frontier at ***iteration i*** is ***piece i*** of vertices in queue
- DFS stack is implicit
  - It's the call stack of the python interpreter
  - From v, recurse on one child at a time
  - But same order if put all children on stack, then pull off (and recurse) one at a time

# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS). DFS vs. BFS***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Topological Sort

# Job Scheduling

- Given
  - A set of tasks
  - Precedence constraints
    - saying “u must be done before v”
  - Represented as a directed graph
- Goal:
  - Find an ordering of the tasks that satisfies all precedence constraints

# Scheduling a set of jobs

Make bus in seconds flat

Fall out of bed

Drag a comb across my head

Look up (at clock)

Find my coat

Notice that I'm late

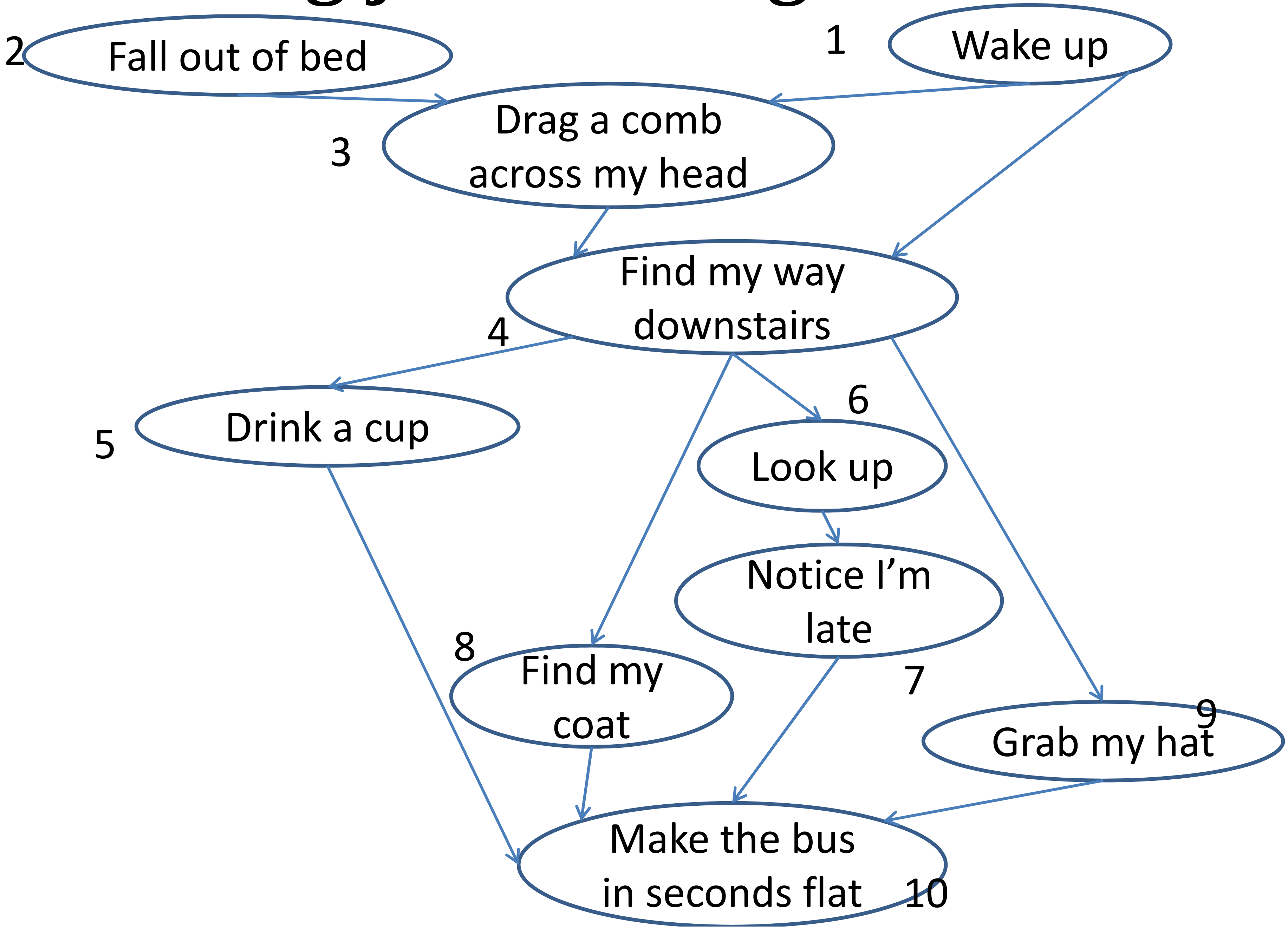
Drink a cup

Wake up

Find my way downstairs

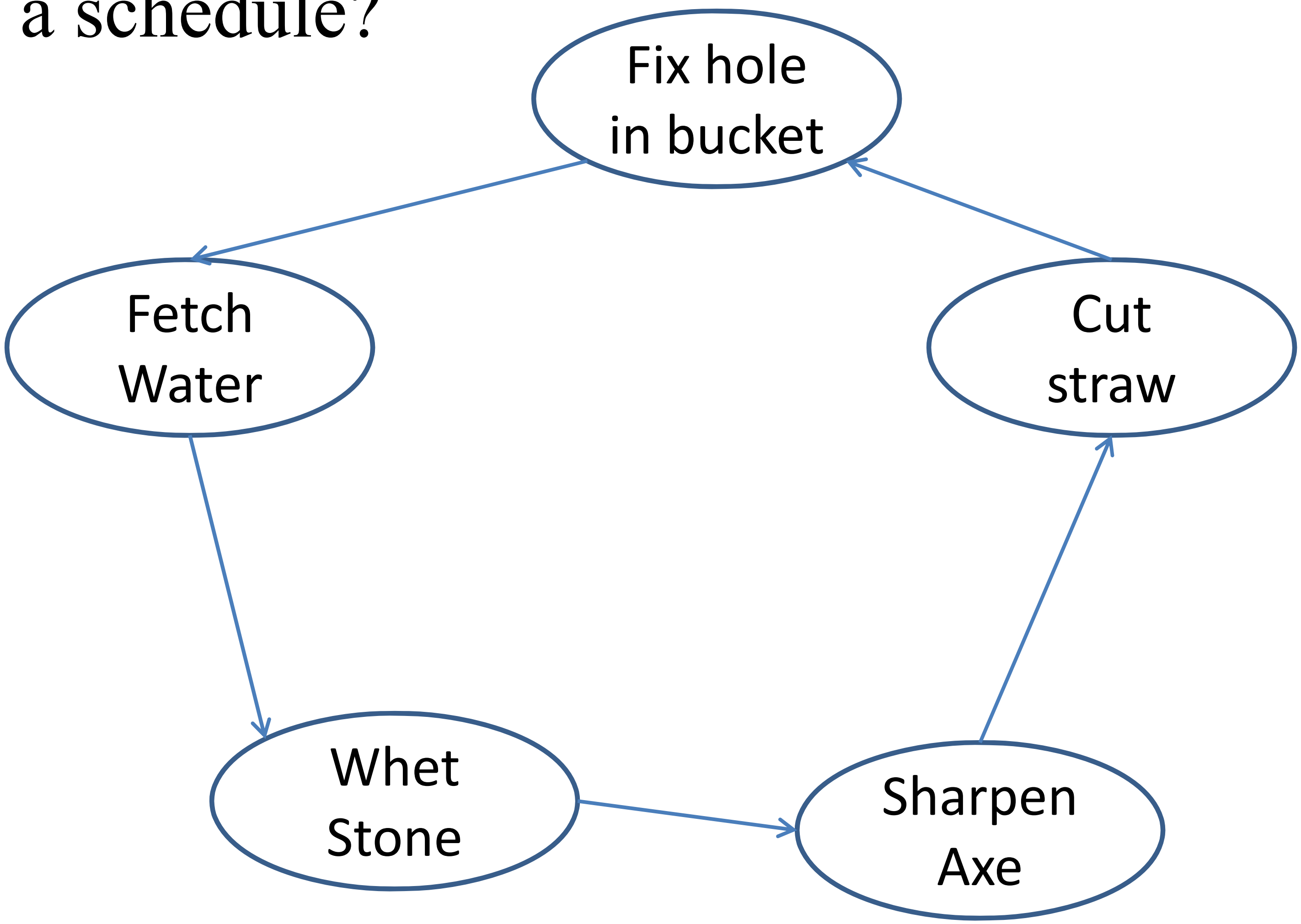
Grab my hat

# Defining job ordering constraints



# Feasibility / schedule existence

- Is there a schedule?




- Each requires previous one to be completed first

# Directed Acyclic Graphs (DAGs)

- Directed Acyclic Graph
  - Graph with no cycles → A schedule exists!
- Source: vertex with no incoming edges
- Claim: every DAG has a source
  - Start anywhere, follow edges backwards
  - If never get stuck, must repeat vertex
  - So, get stuck at a source
- Conclude: every DAG has a schedule
  - Find a source, it can go first
  - Remove, schedule rest of work recursively

# Scheduling algorithm 1 (for DAGs)

- Find a source
  - Scan vertices to find one with no incoming edges
  - Or use DFS on backwards graph
- Remove, recurse
- Time to find one source
  - O(m) with standard adjacency list representation
  - Scan all edges, count occurrence of every vertex as tail
- Total: O(nm)

# Scheduling algorithm 2 (for DAGs)

- Consider DFS
- Observe that we don't return from recursive call to DFS(v) until all of v's children are finished
- So, "finish time" of v is later than finish time of all children
- Thus, later than finish time of all descendants
  - i.e., vertices reachable from v
  - Descendants well-defined since no cycles
- So, reverse of finish times is valid schedule

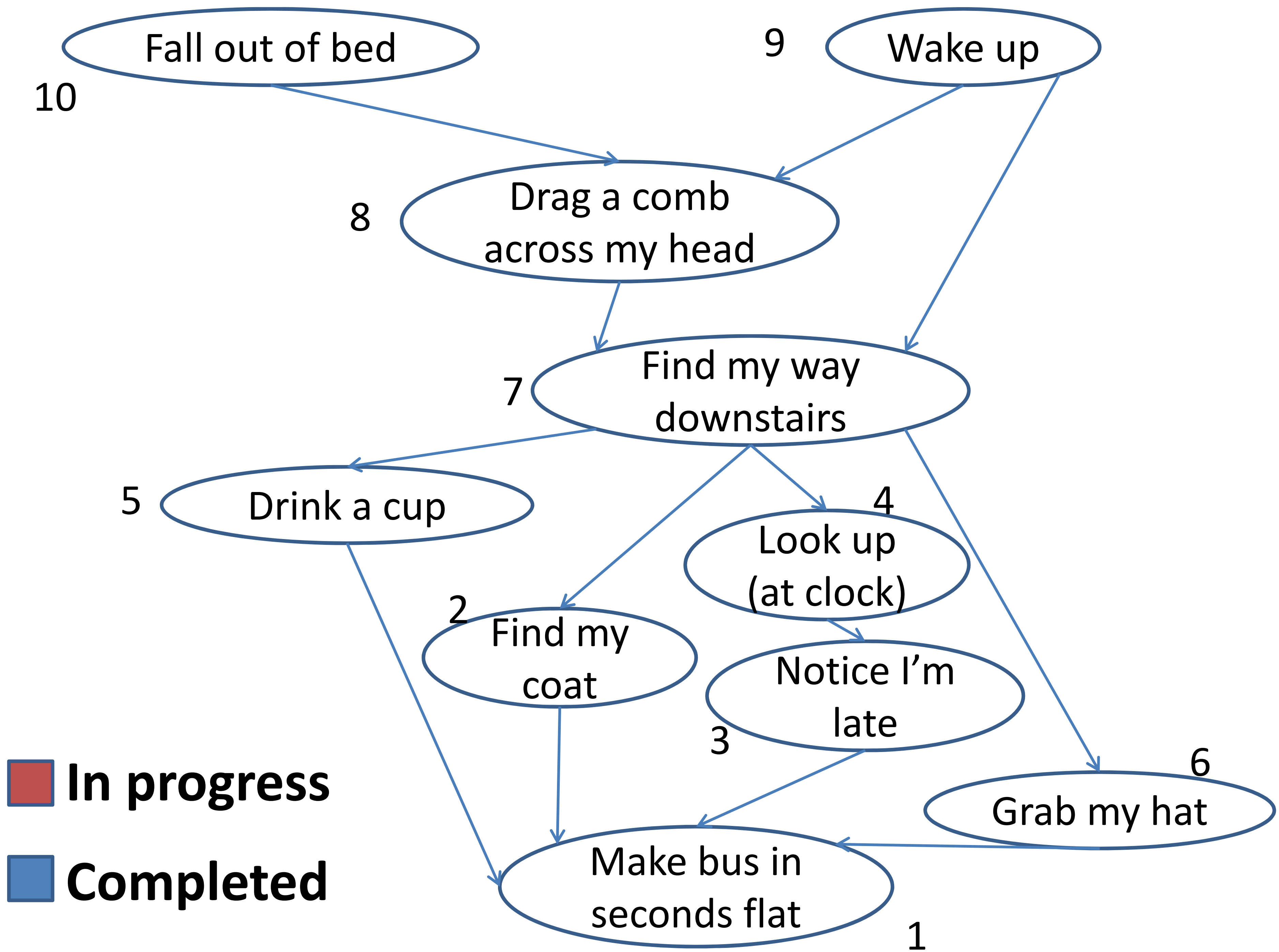
# Implementation of scheduling alg 2

- ***seen*** = {}; ***finishes*** = {}; ***time*** = 0

  DFS-visit (s)
      for v in Adj[s]
          if v not in ***seen***
              ***seen***[v] = 1
              DFS-visit (v)
              ***time*** = ***time***+1
              ***finishes***[v] = ***time***

*only set **finishes** if done processing all edges leaving v*

- TopologicalSort
      for s in V
          DFS-visit(s)
- Sort vertices by ***finishes***[] key


Fall out of bed
10
9
Wake up
8
Drag a comb across my head
7
Find my way downstairs
5
Drink a cup
4
Look up (at clock)
2
Find my coat
Notice I'm late
3
6
Grab my hat
Make bus in seconds flat
1
In progress
Completed

# Analysis

- Just like connected components DFS
  - Time to DFS-Visit from all vertices is O(m+n)
  - Because we do nothing with already seen vertices
- Might DFS-visit a vertex v before its ancestor u
  - i.e., start in middle of graph
  - Does this matter?
  - No, because finish[v] < finish[u] in that case

# Handling Cycles

- If two jobs can reach each other, we must do them at same time
- Two vertices are strongly connected if each can reach the other
- Strongly connected is an equivalence relation
  - So graph has strongly connected components
- Can we find them?
  - Yes, another nice application of DFS
  - But tricky (see CLRS)
  - You should understand algorithm, not proof

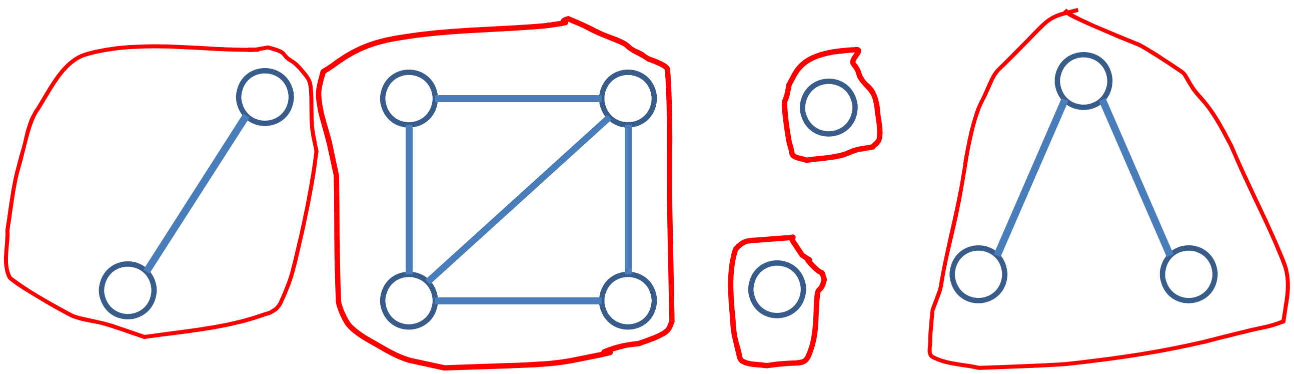
# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS).***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Connected Components

# Connected Components

- Undirected graph G=(V,E)
- Two vertices are connected if there is a path between them
- An equivalence relation
- Equivalence classes are called components
  - A set of vertices all connected to each other



# Finding all connected components

**To find one connected component:**

- The key idea: Both DFS and BFS will reach all vertices reachable from starting vertex s
  - i.e., the ‘component’ of any starting vertex s
- Start with any vertex s:
  - Run DFS (or BFS) to find all vertices in component
  - Mark them as belonging to the same component as s

**To find all connected components:**

- Run the above search *n* times
  - Starting with every vertex

# Naïve Algorithm: DFS *n* times

- DFS-visit (u, ***owner***, o)
```
    #mark all nodes reachable from u with owner o
    for v in Adj[u]
        if v not in owner          #not yet seen
            owner[v] = o           #instead of parent
            DFS-visit (v, owner, o)
```
- DFS-Visit(s, owner, s) will mark owner[v]=s for any vertex reachable from s

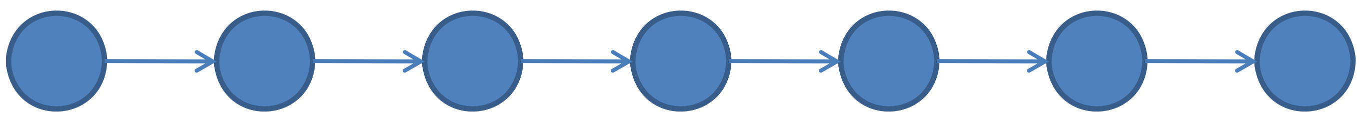
- Correctness:
  - All vertices in same component will receive the same ownership labels
- Cost?
  - n times BFS/DFS? → O(n(m+n))?

# Better: DFS only for unmarked vertices

- If vertex has already been reached, don't need to search from it!
  - Its connected component already marked with owner
- ***owner*** = {}    # global variable owner
  for s in V
      if not(s in ***owner***)
          DFS-Visit(s, ***owner***, s)   #or can use BFS
- Now every vertex examined exactly twice
  - Once in outer loop and once in DFS-Visit
- And every edge examined once
  - In DFS-Visit when its tail vertex is examined
- Total runtime to find components is O(m+n)

# Directed Graphs

- In undirected graphs, connected components can be represented in n space
  - One "owner label" per vertex
- Can ask to compute all vertices reachable from each vertex in a directed graph
  - i.e. the "transitive closure" of the graph
  - Answer can be different for each vertex
  - Explicit representation may be bigger than graph
  - E.g. size n graph with size $n^2$ transitive closure



# Goal for today: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS).***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Global properties of networks

Mostly pointers for further reading

# Networks in the real world

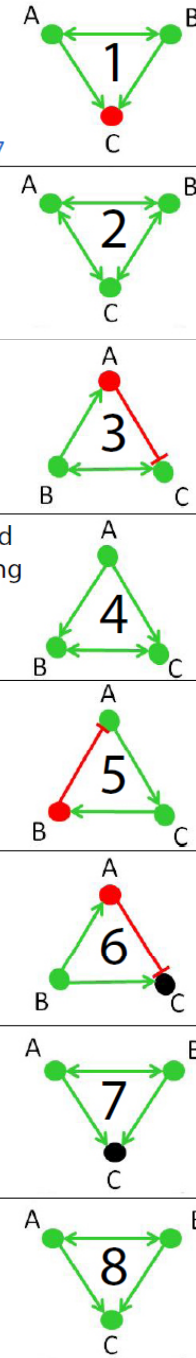

- ***Infrastructure***: Internet, power, transport, distribution
- ***Social***: friends, actors, co-authors, affiliation members
- ***Information***: web pages, paper citations, patents, file-sharing, shopping lists, document-keyword
- ***Biology***: physical, metabolic, regulatory, neural, ecological

# Properties of real-world networks

- ***Small-world property:*** Milgram 6-degrees ('60s)
  - Any pair of vertices connected by short paths
  - People *find* these paths with no global information
- ***'Scale-free'/power-law degree distribution:***
  - 80/20 rule: 80% of connections in 20% of vertices
  - Few heavily-connected hubs, most lie in the fringes
- ***Network growth*** and ***preferential attachment***
  - Rich-get-richer can lead to power-law distributions
- ***Clustering coefficient***: average probability that ***v***'s neighbors are also connected to each other.
  - Measures the density of closed vs. open 'triangles'
  - More generally: measure frequency of all ***network motifs***, i.e. over-/under-representation of all sub-graphs size 3,4,5,…

# Network ‘motifs’

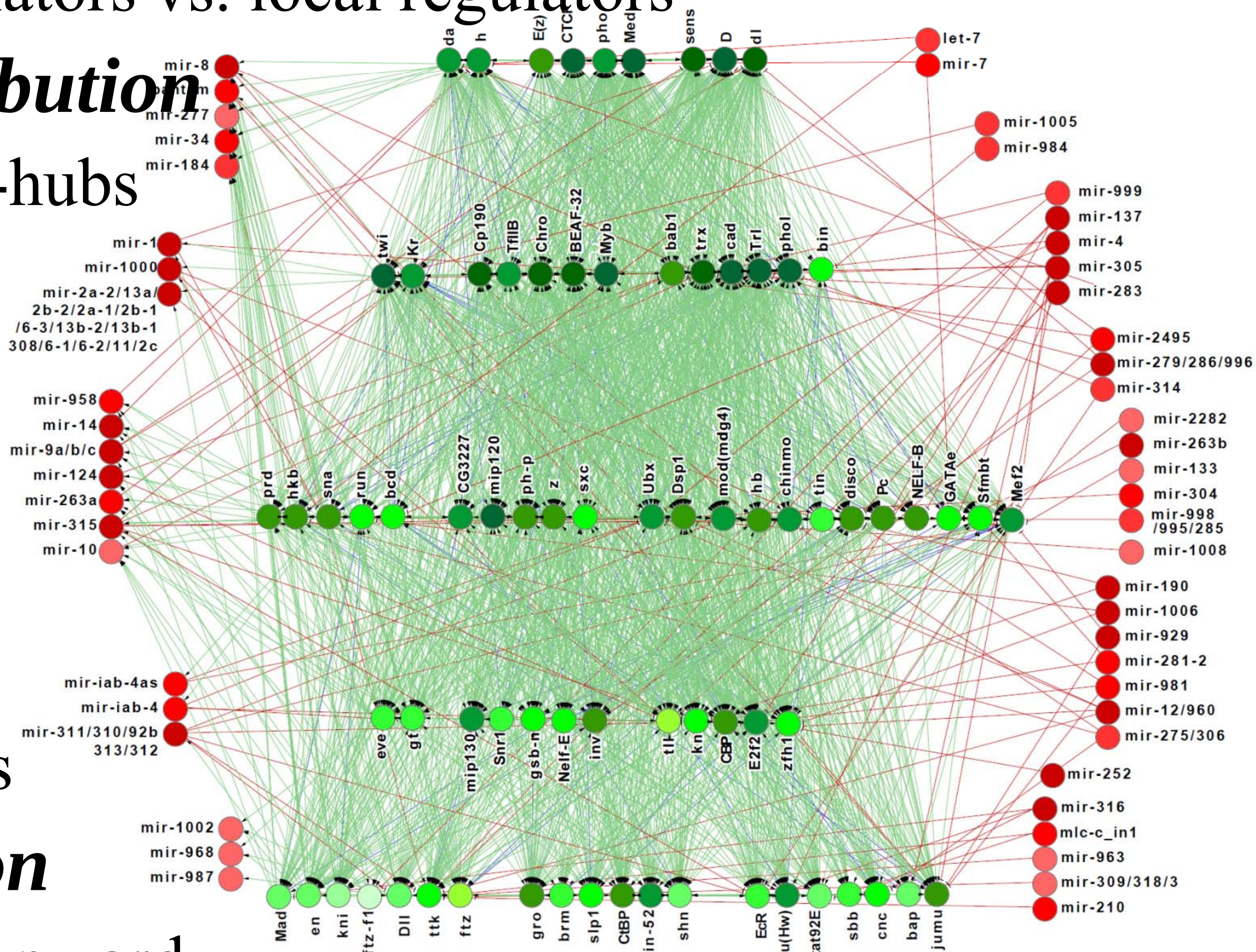
- **Network building blocks**
  - Smallest meaningful unit
- **Interpretable circuit components**
  - Feed-forward loops
  - Feedback loops
  - Cross-regulation
  - Amplification, etc
- **Discovered based on their over-representation**
  - Compared to ‘random’ net




| # | Network Motif | Fold Enr. | Z-score | Example genes A | Example genes B | Example genes C |
|---|---|---|---|---|---|---|
| 1 | Cross-regulating TFs co-targeting a miRNA | F=2.996 | Z=5.637 | twi, sna, eve, run, kni | h, prd, gt, prd, gt | bantam, mir-8, mir-10, mir-14, mir-277 |
| 2 | Cross-regulatory clique of TFs | 2.063 | 3.453 | bcd, mod(mdg4), BEAF-32, Cp190, cad | cad, Kr, Myb, mip120, Chro | ph-p, Dsp1, Med, phol, dl |
| 3 | Feed-forward loop with cross-regulating TFs and a miRNA | 1.969 | 2.311 | mir-1, mir-315, mir-14, mir-263a, mir-8 | twi, gt, run, prd, h | sna, Kr, h, Kr, hb |
| 4 | Double feed-forward loop: cross-regulating TFs co-targeted by another TF | 1.294 | 4.507 | prd, bab1, TfIIB, da, GATAe | gt, trx, mip130, Mef2, Mef2 | shn, disco, Myb, lin-52, z |
| 5 | Feedback loop from downstream TF to upstream TF via a microRNA | 1.273 | 1.295 | tin, sna, Kr, hb, sens | mir-1000, mir-1, mir-315, mir-8, mir-9abc | Kr, C15, sna, nub, eve |
| 6 | Feed-forward loop with a miRNA ending at a target gene | 1.259 | 1.797 | mir-958, bantam, mir-8, mir-124, mir-263a | hkb, twi, sna, sna, run | Csk, dap, crb, Gli, Mes2 |
| 7 | Cross-regulating TFs co-targeting a target gene | 1.256 | 12.625 | pho, D, dl, phol, mip120 | kn, da, lin-52, Med, Myb | Keap1, dnk, px, tna, Moe |
| 8 | Cross-regulating TFs co-targeting another TF | 1.158 | 7.117 | z, Dsp1, gt, trx, prd | Med, phol, shn, disco, ph-p | run, lin-52, cic, CBP, Antp |

# Interpreting biological network properties

- ***Hierarchical organization***
  - Master regulators vs. local regulators
- ***Degree distribution***
  - In-hubs, out-hubs
- ***Diameter***
  - Info transfer
- ***Modularity***
  - Locality
- ***Clustering***
  - Subnetworks
- ***Flow direction***
  - Downward/upward




e.g. modENCODE consortium, <u>Science</u>, 2010

# Node properties: Centrality (hubs)

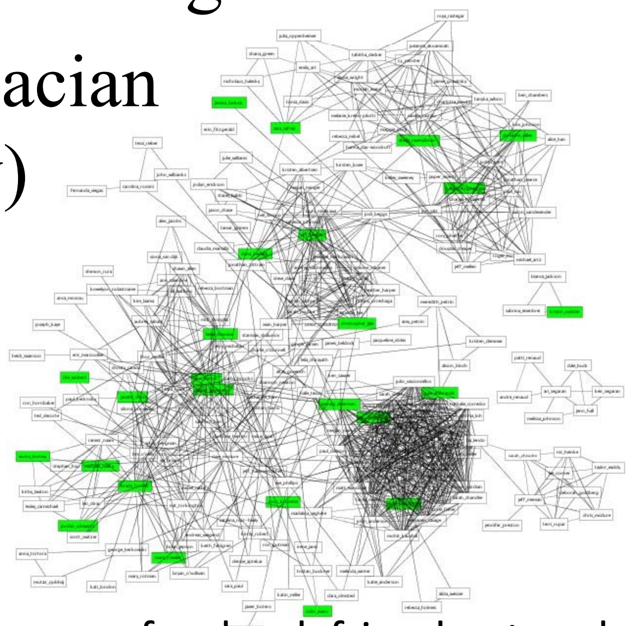
- Centrality of node ***v*** can be measured as:
  1. ***Degree centrality***: Number of in/out-edges for ***v***, i.e. number of neighbors as measure of importance/authority.
  2. ***Eigenvector centrality***: sum of centrality of ***v***'s neighbors; high when ***v*** has many neighbors or 'central' neighbors
  3. ***Katz centrality***: balances 1 (# of neighbors) and 2 (neighbor centrality) using a weighting parameter
  4. ***Page rank***: dilutes 'centrality' flow out of a vertex by its number of neighbors. Used in Google search results.
  5. ***Closeness centrality***: mean distance to other vertices.
  6. ***Betweeness centrality:*** # of shortest paths through ***v***.
  7. ***Flow-betweeness***: amount of flow through ***v*** for all (s,t)
  8. ***Random-walk betweeness***: s diffusion, sink t, traversing ***v***

# Node pairs: Similarity/Closeness

- ***Assortative mixing:*** Nodes with similar properties are similar, in the same component, clique, etc…
- ***Node similarity***, *or* ***node equivalence***:
  - ***Structural***: share many of the same neighbors
  - ***Regular***: share neighbors with similar properties
- ***Property clustering:*** A set of ***n*** nodes can form a:
  - ***Clique***: fully connected, each ***n***-1 neighbors
  - ***k-plex:*** nearly fully connected, each ***n-k*** neighbors
  - ***k-core:*** each ***k*** neighbors. Note: *k*-core=(*n*-*k*)-plex
- Defining graph neighborhoods with ***components***:
  - ***Component***: Any 2 nodes linked by at least one path
  - ***k*-component**: at least ***k*** vertex-independent paths

# Beyond components / k-components

- Many networks have 1 giant connected component
  - But sub-structure exists within it eg.‘clusters’ of friends
- ***Graph partitioning*** algorithms. Break into k clusters
  - Simplest form: ***graph bisection*** problem. NP complete
    - Exhaustive search $(2^{\boldsymbol{n}+1})/\sqrt{\boldsymbol{n}}$ partitions. Only heuristics
  - Kernigan-Lin: Divide randomly, and re-assign members
  - Spectral partitioning: uses graph Laplacian measures ‘diffusion’ (vs. connectivity)
- ***Community detection*** algorithms
  - Discover coherent small groups
  - Modularity maximization
    - Spectral, betweeness-based, other

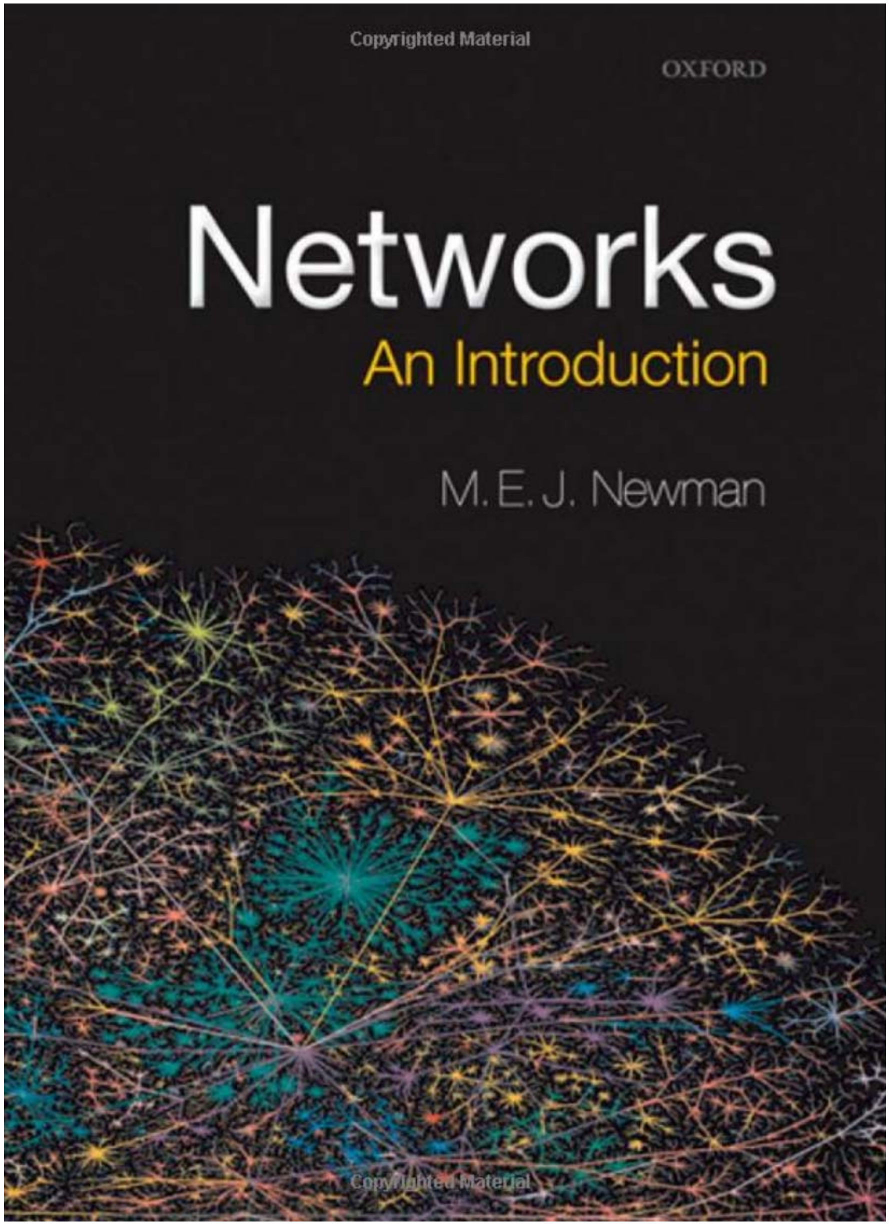


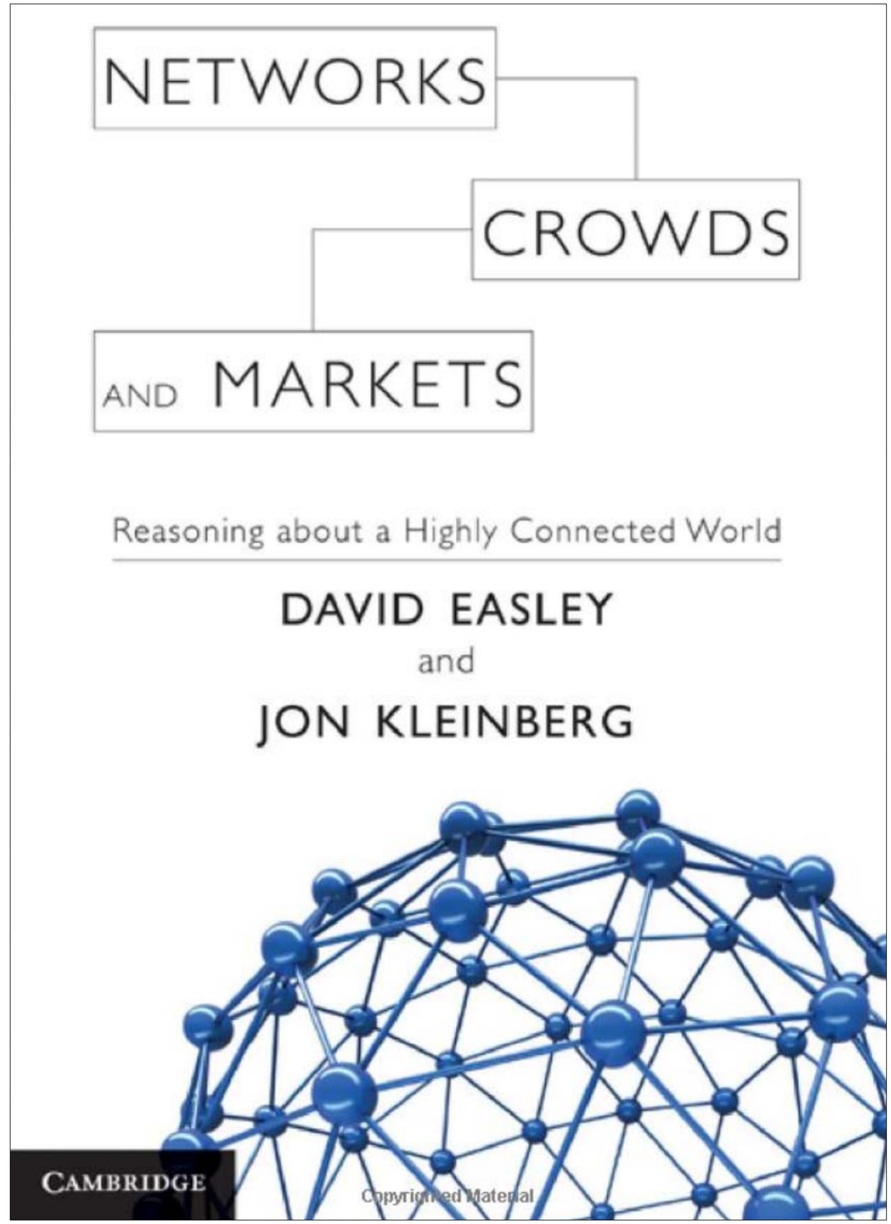
e.g. facebook friend network

# Dynamic processes on networks

- ***Percolation and network resilience***
  - Uniform/non-uniform removal of vertices/edges/hubs
  - E.g. router failure, network attack, vaccination
- ***Epidemics on networks***
  - Spread of disease, susceptible/infected/recovered
  - Time-dependent properties of disease spreading
- ***Dynamical systems on networks, rates, dx/dt***
  - Metabolic modeling, steady-state analysis/fixed points
  - Information flow, stability, synchronization
- ***Network search***
  - Web search, distributed databases, message passing

# Recommended further reading



# Today's recap: Graphs III

- ***Recap on graphs, games, searching, BFS***
  - Defs, Rubik, BFS, correctness, shortest paths
- ***Depth first search (DFS).***
  - Algorithm, runtime, correctness, edge classes
- ***Applications of DFS***
  - Topological Sort on DAGs, job scheduling
  - Connected components, strongly connected
- ***Properties of real-world & biological networks***
  - Types, small-world, scale-free, growth, motifs, interpreting, centrality, similarity, dynamics

# Games, Graphs, Searching, Networks

**Graphs I: Introduction to Games and Graphs**

- Rubik's cube, Pocket cube, Game space
- Graph definitions, representation, searching

**Graphs II: Graph algorithms and analysis**

- Breadth First Search, Depth First Search
- Queues, Stacks, Augmentation, Topological sort

**Graphs III: Networks in biology and real world**

- Network/node properties, metrics, motifs, clusters
- Dynamic processes, epidemics, growth, resilience

# Next: Shortest paths… Happy Spring Break!

| Unit | Pset | Week | Date | | Lecture (Tuesdays and Thursdays) | | Recitation (Wed and Fri) |
|---|---|---|---|---|---|---|---|
| Intro | PS1 | 1 | Tue Feb 01 | 1 | Introduction and Document Distance | 1 | Python and Asymptotic Complexity |
| Binary | Out: 2/1 | | Thu Feb 03 | 2 | Peak Finding Problem | 2 | Peak Finding correctness & analysis |
| Search | Due: Mon 2/14 | 2 | Tue Feb 08 | 3 | Scheduling and Binary Search Trees | 3 | Binary Search Tree Operations |
| Trees | HW lab: Sun 2/13 | | Thu Feb 10 | 4 | Balanced Binary Search Trees | 4 | Rotations and AVL tree deletions |
| Hashing | PS2 Out: 2/15 | 3 | Tue Feb 15 | 5 | Hashing I : Chaining, Hash Functions | 5 | Hash recipes, collisions, Python dicts |
| | Due: Mon 2/28 | | Thu Feb 17 | 6 | Hashing II : Table Doubling, Rolling Hash | 6 | Probability review, Pattern matching |
| | HW lab:Sun 2/27 | 4 | Tue Feb 22 | - | President's Day - Monday Schedule - No Class | - | No recitation |
| | | | Thu Feb 24 | 7 | Hashing III : Open Addressing | 7 | Universal Hashing, Perfect Hashing |
| Sorting | PS3. Out: 3/1 | 5 | Tue Mar 01 | 8 | Sorting I : Insertion & Merge Sort, Master Theorem | 8 | Proof of Master Theorem, Examples |
| | Due: Mon 3/7 | | Thu Mar 03 | 9 | Sorting II : Heaps | 9 | Heap Operations |
| | HW lab: Sun 3/6 | 6 | Tue Mar 08 | 10 | Sorting III: Lower Bounds, Counting Sort, Radix Sort | 10 | Models of computation |
| | | | Wed Mar 09 | Q1 | Quiz 1 in class at 7:30pm. Covers L1-R10. Review Session on Tue 3/8 at 7:30pm. | | |
| Graphs | PS4. Out: 3/10 | | Thu Mar 10 | 11 | Searching I: Graph Representation, Depth-1st Search | 11 | Strongly connected components |
| and | Due: Fri 3/18 | 7 | Tue Mar 15 | 12 | Searching II: Breadth-1st Search, Topological Sort | 12 | Rubik's Cube Solving |
| Search | HW lab:W 3/16 | | Thu Mar 17 | 13 | Searching III: Games, Network properties, Motifs | 13 | Subgraph isomorphism |
| Shortest | PS5 | 8 | Tue Mar 29 | 14 | Shortest Paths I: Introduction, Bellman-Ford | 14 | Relaxation algorithms |
| Paths | Out: 3/29 | | Thu Mar 31 | 15 | Shortest Paths II: Bellman-Ford, DAGs | 15 | Shortest Path applications |
| | Due: Mon 4/11 | 9 | Tue Apr 05 | 16 | Shortest Paths III: Dijkstra | 16 | Speeding up Dijkstra's algorithm |
| | HW lab:Sun 4/10 | | Thu Apr 07 | 17 | Graph applications, Genome Assembly | 17 | Euler Tours |
| Dynamic | PS6 | 10 | Tue Apr 12 | 18 | DP I: Memoization, Fibonacci, Crazy Eights | 18 | Limits of dynamic programming |
| Program | Out: Tue 4/12 | | Wed Apr 13 | Q2 | Quiz 2 in class at 7:30pm. Covers L11-R17. Review Session on Tue 4/13 at 7:30pm. | | |
| ming | Due: Fri 4/29 | | Thu Apr 14 | 19 | DP II: Shortest Paths, Genome sequence alignment | 19 | Edit Distance, LCS, cost functions |
| | HW lab:W 4/27 | 11 | Tue Apr 19 | - | Patriot's Day - Monday and Tuesday Off | - | No recitation |
| | | | Thu Apr 21 | 20 | DP III: Text Justification, Knapsack | 20 | Saving Princess Peach |
| | | 12 | Tue Apr 26 | 21 | DP IV: Piano Fingering, Vertex Cover, Structured DP | 21 | Phylogeny |
| Numbers | PS7 out Thu4/28 | | Thu Apr 28 | 22 | Numerics I - Computing on large numbers | 22 | Models of computation return! |
| Pictures | Due: Fri 5/6 | 13 | Tue May 3 | 23 | Numerics II - Iterative algorithms, Newton's method | 23 | Computing the nth digit of $\pi$ |
| (NP) | HW lab: Wed 5/4 | | Thu May 5 | 24 | Geometry: Line sweep, Convex Hull | 24 | Closest pair |
| | | 14 | Tue May 10 | 25 | Complexity classes, and reductions | 25 | Undecidability of Life |
| Beyond | | | Thu May 12 | 26 | Research Directions (15 mins each) + related classes | | |
| | | 15 | Finals week | Q3 | Final exam is cumulative L1-L26. Emphasis on L18-L26. Review Session on Fri 5/13 at 3pm | | |