Wrap-Up Hodgepodge

CSE 332 Sp25
Lecture 28

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week			Ex 13 (MST,prog) due		TODAY Ex 14 (P/NP, gs) due
Next Week				Final :O	

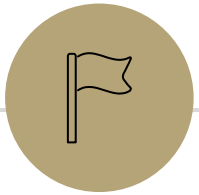
Outline

3 topics:

Topological Sort

Strongly Connected Components

P/NP wrap-up



Two More Simple Graph Algorithms

What you need to know

What is a topological sort and a DAG?

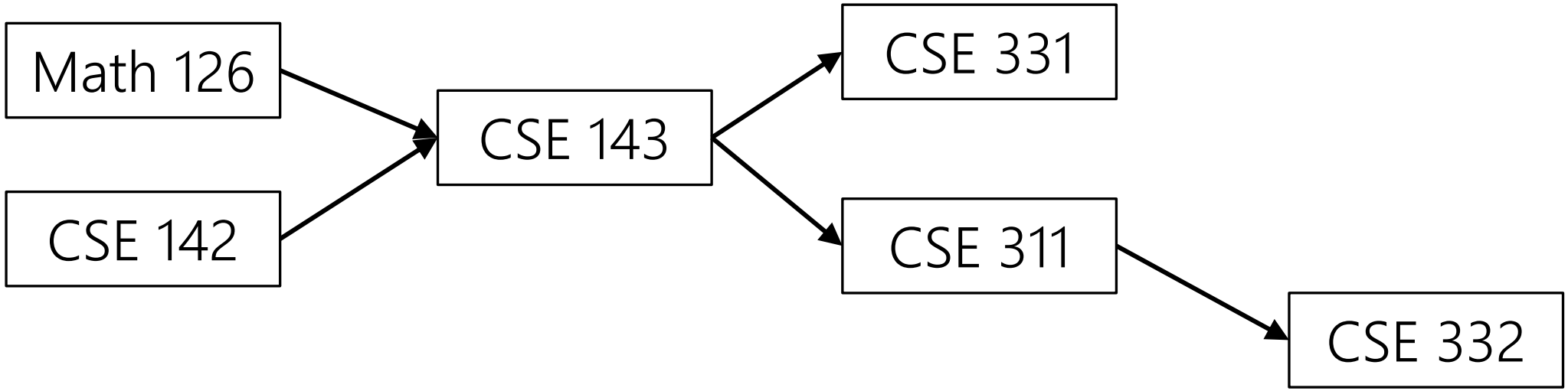
When does a topological sort exist and when doesn't it?

A topological sort can be found in O(V+E) time

Be able to find a topological sort yourself (by hand, don't need to show separate algorithm steps).

Ordering Dependencies

Today's next problem: Given a bunch of courses with prerequisites, find an order to take the courses in.



Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

Can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)

Given: a directed graph G

Find: an ordering of the vertices so all edges go from left to right.

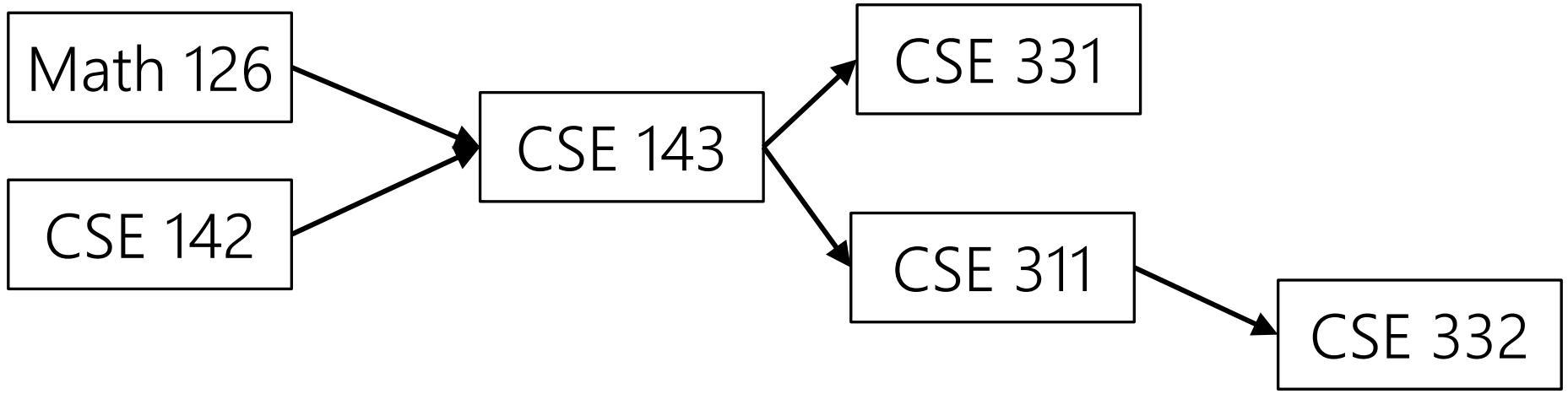
Uses:

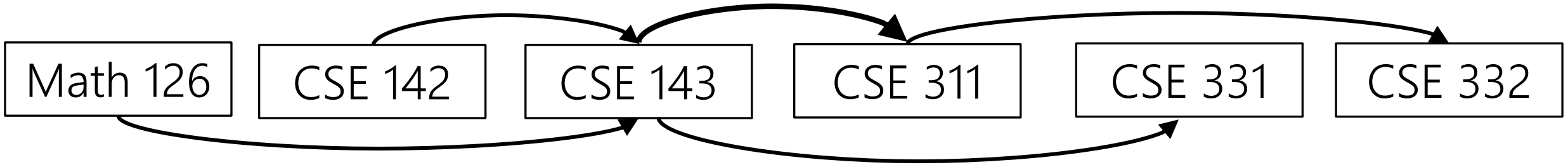
Compiling multiple files

Graduating.

Topological Ordering

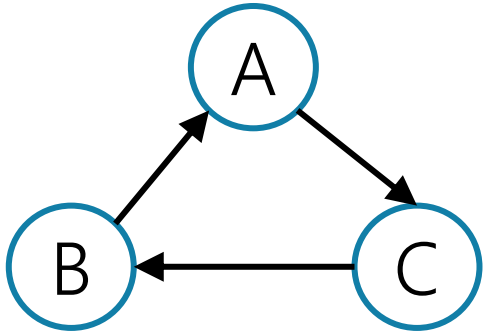
A course prerequisite chart and a possible topological ordering.





Can we always order a graph?

Can you topologically order this graph?



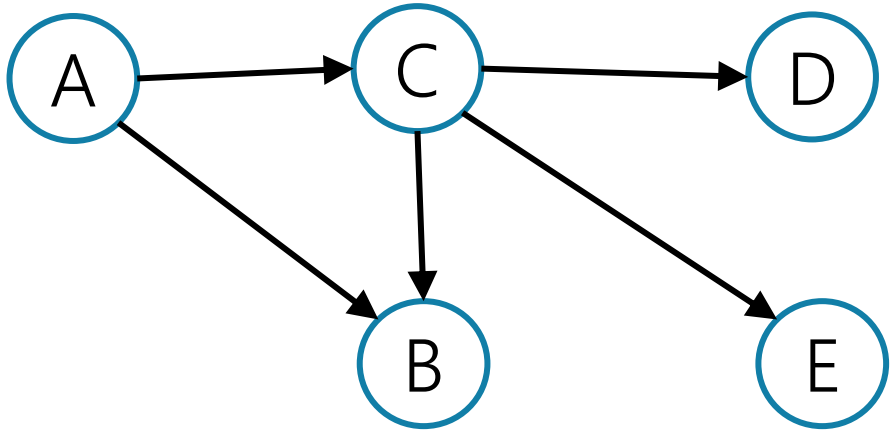
Directed Acyclic Graph (DAG)

A directed graph without any cycles.

A graph has a topological ordering if and only if it is a DAG.

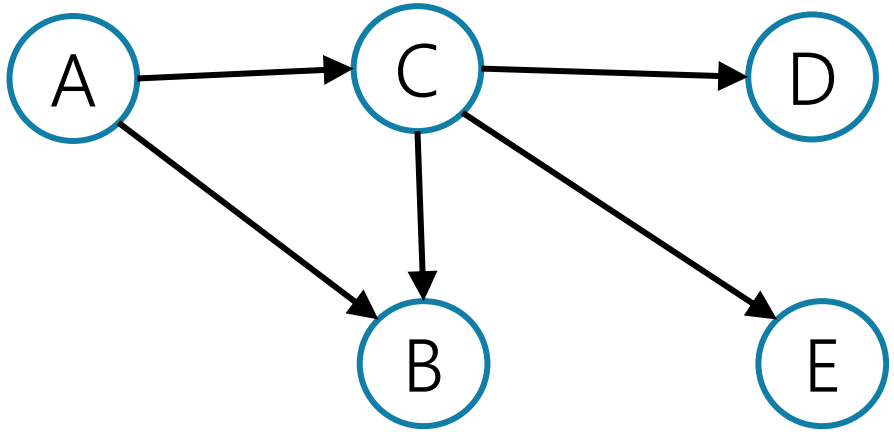
Ordering a DAG

Does this graph have a topological ordering? If so find one.



Ordering a DAG

Does this graph have a topological ordering? If so find one.



If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

How Do We Find a Topological Ordering?

```
TopologicalSort(Graph G, Vertex source)
   count how many incoming edges each vertex has
       Collection toProcess = new Collection()
       foreach(Vertex v in G){
              if(v.edgesRemaining == 0)
                     toProcess.insert(v)
   }
       topOrder = new List()
       while(toProcess is not empty){
              u = toProcess.remove()
              topOrder.insert(u)
              foreach(edge (u,v) leaving u){
                     v.edgesRemaining--
                     if(v.edgesRemaining == 0)
                            toProcess.insert(v)
              }
       }
```

What's the running time?

```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
        Collection toProcess = new Collection()
        foreach(Vertex v in G){
                if(v.edgesRemaining == 0)
                        toProcess.insert(v)
    }
        topOrder = new List()
        while(toProcess is not empty){
                u = toProcess.remove()
                topOrder.insert(u)
                foreach(edge (u,v) leaving u){
                        v.edgesRemaining--
                        if(v.edgesRemaining == 0)
                                toProcess.insert(v)
                }
        }
```

Running Time: $O(|V| + |E|)$

Finding a Topological Ordering

Instead of counting incoming edges, you can actually modify DFS to find you one (think about why).

But the “count incoming edges” is a bit easier to understand (for me ☺)

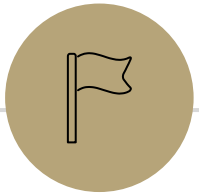
What you need to know

What is a topological sort and a DAG?

When does a topological sort exist and when doesn't it?

A topological sort can be found in O(V+E) time

Be able to find a topological sort yourself (by hand, don't need to show algorithm steps).



Connectivity in Directed Graphs

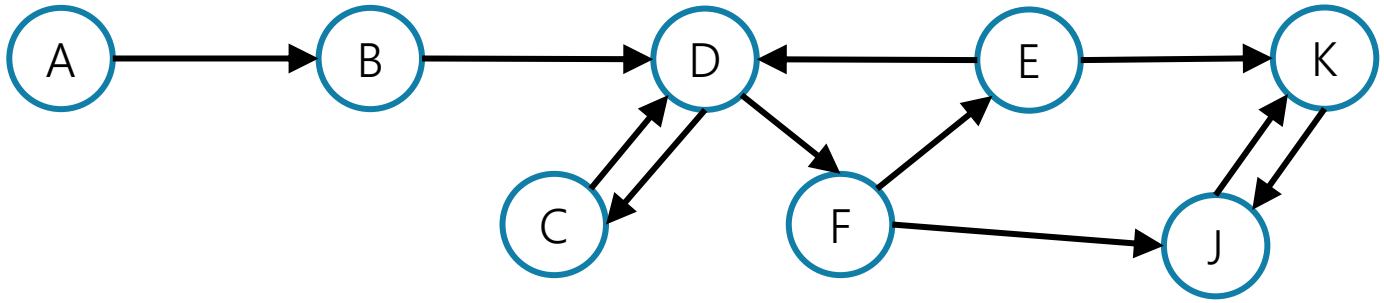
What you need to know

Definition of strongly connected

Strongly connected components can be found in O(V+E) time via modification of DFS

Find SCCs of a graph (by hand, don’t need to show algorithm steps)

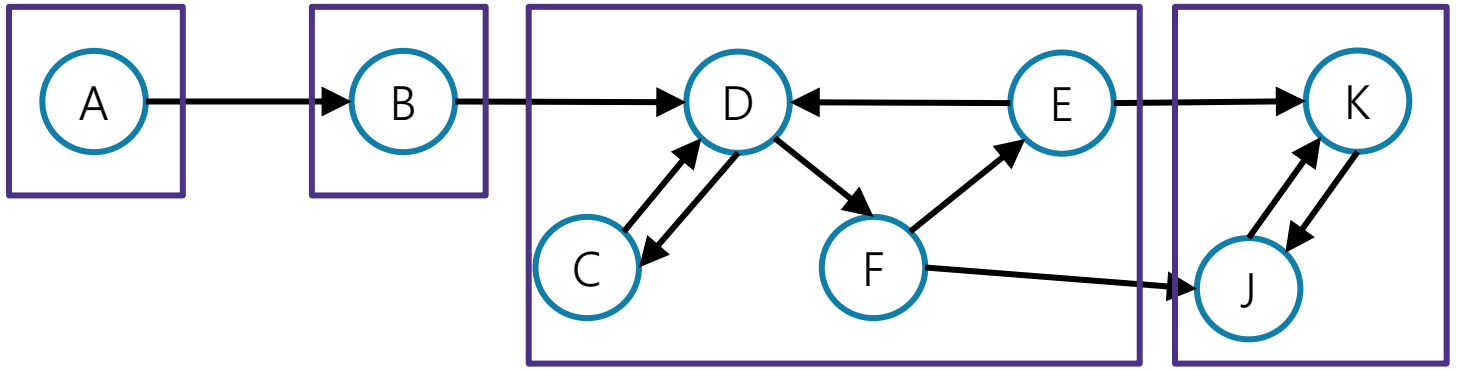
Problem 2: Find Strongly Connected Components



Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.

Problem 2: Find Strongly Connected Components



{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.

Connectedness Definitions

In an undirected graph, a connected component is a “piece” of the graph: a vertex and everything its connected to via a path.

Equivalently, a subgraph C such that every pair of vertices in C is connected via some path and there is no other vertex which is connected to every vertex of C in both directions.

In a directed graph, you might care about

Weakly connected components (ignore the directions on the edges, if it were undirected, would it be connected?)

Strongly connected (can you get in both directions)

Can you find Strongly Connected Components?

A couple of different ways to use DFS to find strongly connected components.

Wikipedia has the details.

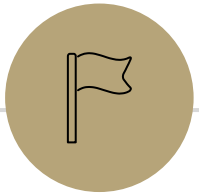
High level: need to keep track of “highest point” in DFS tree you can reach back up to.

What you need to know

Definition of strongly connected

Strongly connected components can be found in O(V+E) time via modification of DFS

Find SCCs of a graph (by hand, don’t need to show algorithm steps)



Why P vs. NP matters

P (stands for "Polynomial")

The set of all decision problems that have an algorithm that runs in time $O(n^k)$ for some constant k.

NP (stands for "nondeterministic polynomial")

The set of all decision problems such that if the answer is YES, there is a proof of that which can be verified in polynomial time.

NP-complete

Problem B is NP-complete if B is in NP and
for all problems A in NP, A reduces to B in polynomial time.

NP-hard

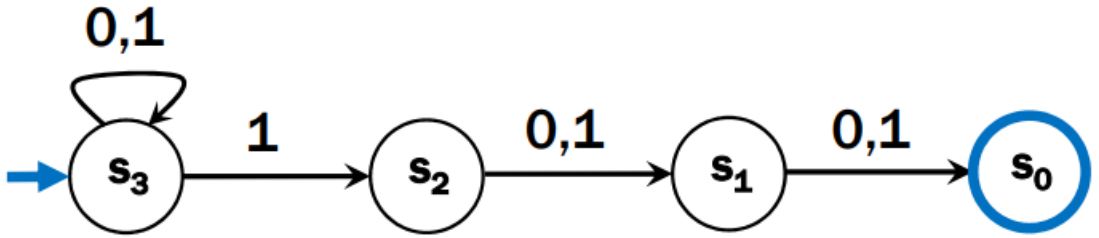
Problem B is NP-hard if
for all problems A in NP, A reduces to B in polynomial time.

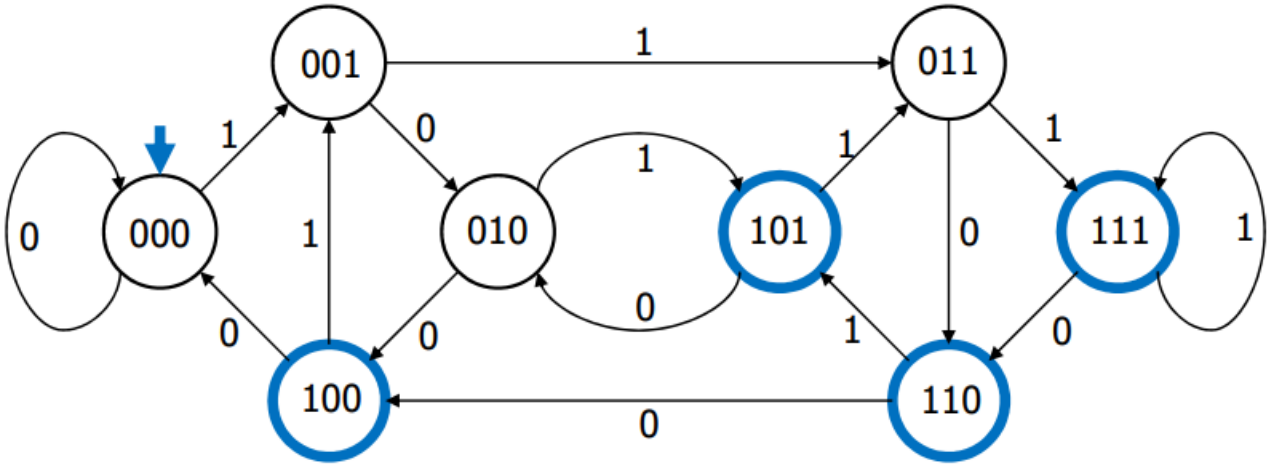
Why is it called NP?

You've seen nondeterministic computation before.
Back in 311.

NFAs would "magically" decide among a set of valid transitions.
Always choosing one that would lead to an accept state (if such a transition exists).

An NFA and a DFA for the language "binary strings with a 1 in the 3rd position from the end."





From Kevin & Paul's 311 Lecture 23.

Nondeterminism

What would a nondeterministic **computer** look like?

It can run all the usual commands,

But it can also magically (i.e. nondeterministically) decide to set any bit of memory to 0 or 1.

Always choosing 0 or 1 to cause the computer to output YES,

(if such a choice exists).

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-COLOR?

Solve 3-COLOR?

If we had a nondeterministic computer…

Can you think of a polynomial time algorithm on a nondeterministic computer to:

Solve 2-COLOR?

Just run our regular deterministic polynomial time algorithm

Or nondeterministically guess colors, output if they work.

Solve 3-COLOR?

nondeterministically guess colors, output if they work.

Analogue of NFA/DFA equivalence

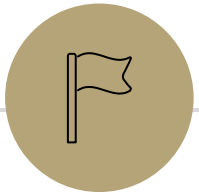
You showed in 311 that the set of languages decided by NFAs and DFAs were the same.

I.e. NFAs didn't let you solve more problems than DFAs.

But it did sometimes make the process a lot easier.

There are languages such that the best DFA is exponentially larger than the best NFA. (like the one from a few slides ago).

P vs. NP is an analogous question. Does non-determinism let us use exponentially fewer resources to solve some problems?



History, and Why P vs. NP?

NP-Completeness

An NP-complete problem is a **universal language** for encoding "I'll know it when I see it" problems.

If you find an efficient algorithm for an NP-complete problem, you have an algorithm for **every** problem in NP

Cook-Levin Theorem (1971)

SAT is NP-complete

Theorem 1: If a set S of strings is accepted by some nondeterministic Turing machine within polynomial time, then S is P-reducible to {DNF tautologies}.

NP-Complete Problems

But Wait! There's more!

Main Theorem. All the problems on the following list are complete.

1. SATISFIABILITY
   COMMENT: By duality, this problem is equivalent to determining whether a disjunctive normal form expression is a tautology.

2. 0-1 INTEGER PROGRAMMING
   INPUT: integer matrix C and integer vector d
   PROPERTY: There exists a 0-1 vector x such that $Cx = d$.

3. CLIQUE
   INPUT: graph G, positive integer k
   PROPERTY: G has a set of k mutually adjacent nodes.

4. SET PACKING
   INPUT: Family of sets $\{S_j\}$, positive integer ℓ
   PROPERTY: $\{S_j\}$ contains ℓ mutually disjoint sets.

5. NODE COVER
   INPUT: graph G', positive integer ℓ
   PROPERTY: There is a set $R \subseteq N'$ such that $|R| \leq \ell$ and every arc is incident with some node in R.

6. SET COVERING
   INPUT: finite family of finite sets $\{S_j\}$, positive integer k
   PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ containing $\leq k$ sets such that $\cup T_h = \cup S_j$.

7. FEEDBACK NODE SET
   INPUT: digraph H, positive integer k
   PROPERTY: There is a set $R \subseteq V$ such that every (directed) cycle of H contains a node in R.

8. FEEDBACK ARC SET
   INPUT: digraph H, positive integer k
   PROPERTY: There is a set $S \subseteq E$ such that every (directed) cycle of H contains an arc in S.

9. DIRECTED HAMILTON CIRCUIT
   INPUT: digraph H
   PROPERTY: H has a directed cycle which includes each node exactly once.

10. UNDIRECTED HAMILTON CIRCUIT
    INPUT: graph G
    PROPERTY: G has a cycle which includes each node exactly once.

11. SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE
    INPUT: Clauses $D_1, D_2, \ldots, D_r$, each consisting of at most 3 literals from the set $\{u_1, u_2, \ldots, u_m\} \cup \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_m\}$
    PROPERTY: The set $\{D_1, D_2, \ldots, D_r\}$ is satisfiable.

12. CHROMATIC NUMBER
    INPUT: graph G, positive integer k
    PROPERTY: There is a function $\phi: N \to Z_k$ such that, if u and v are adjacent, then $\phi(u) \neq \phi(v)$.

13. CLIQUE COVER
    INPUT: graph G', positive integer ℓ
    PROPERTY: N' is the union of ℓ or fewer cliques.

14. EXACT COVER
    INPUT: family $\{S_j\}$ of subsets of a set $\{u_i, i = 1,2,\ldots,t\}$
    PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i, i = 1,2,\ldots,t\}$.

15. HITTING SET
    INPUT: family $\{U_i\}$ of subsets of $\{s_j, j = 1,2,\ldots,r\}$
    PROPERTY: There is a set W such that, for each i, $|W \cap U_i| = 1$.

16. STEINER TREE
    INPUT: graph G, $R \subseteq N$, weighting function $w: A \to Z$, positive integer k
    PROPERTY: G has a subtree of weight $\leq k$ containing the set of nodes in R.

17. 3-DIMENSIONAL MATCHING
    INPUT: set $U \subseteq T \times T \times T$, where T is a finite set
    PROPERTY: There is a set $W \subseteq U$ such that $|W| = |T|$ and no two elements of W agree in any coordinate.

18. KNAPSACK
    INPUT: $(a_1, a_2, \ldots, a_r, b) \in Z^{n+1}$
    PROPERTY: $\sum a_j x_j = b$ has a 0-1 solution.

19. JOB SEQUENCING
    INPUT: "execution time vector" $(T_1, \ldots, T_p) \in Z^p$,
    "deadline vector" $(D_1, \ldots, D_p) \in Z^p$
    "penalty vector" $(P_1, \ldots, P_p) \in Z^p$
    positive integer k
    PROPERTY: There is a permutation π of $\{1,2,\ldots,p\}$ such that

$$\left(\sum_{j=1}^{p} [\text{if } T_{\pi(1)}+\cdots+T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0]\right) \leq k \ .$$

20. PARTITION
    INPUT: $(c_1, c_2, \ldots, c_s) \in Z^s$
    PROPERTY: There is a set $I \subseteq \{1,2,\ldots,s\}$ such that $\sum_{h \in I} c_h = \sum_{h \notin I} c_h$.

21. MAX CUT
    INPUT: graph G, weighting function $w: A \to Z$, positive integer W
    PROPERTY: There is a set $S \subseteq N$ such that

$$\sum_{\substack{\{u,v\} \in A \\ u \in S \\ v \notin S}} w(\{u,v\}) \geq W \ .$$

Karp's Theorem (1972)

A lot of problems people care about are NP-complete

NP-Complete Problems

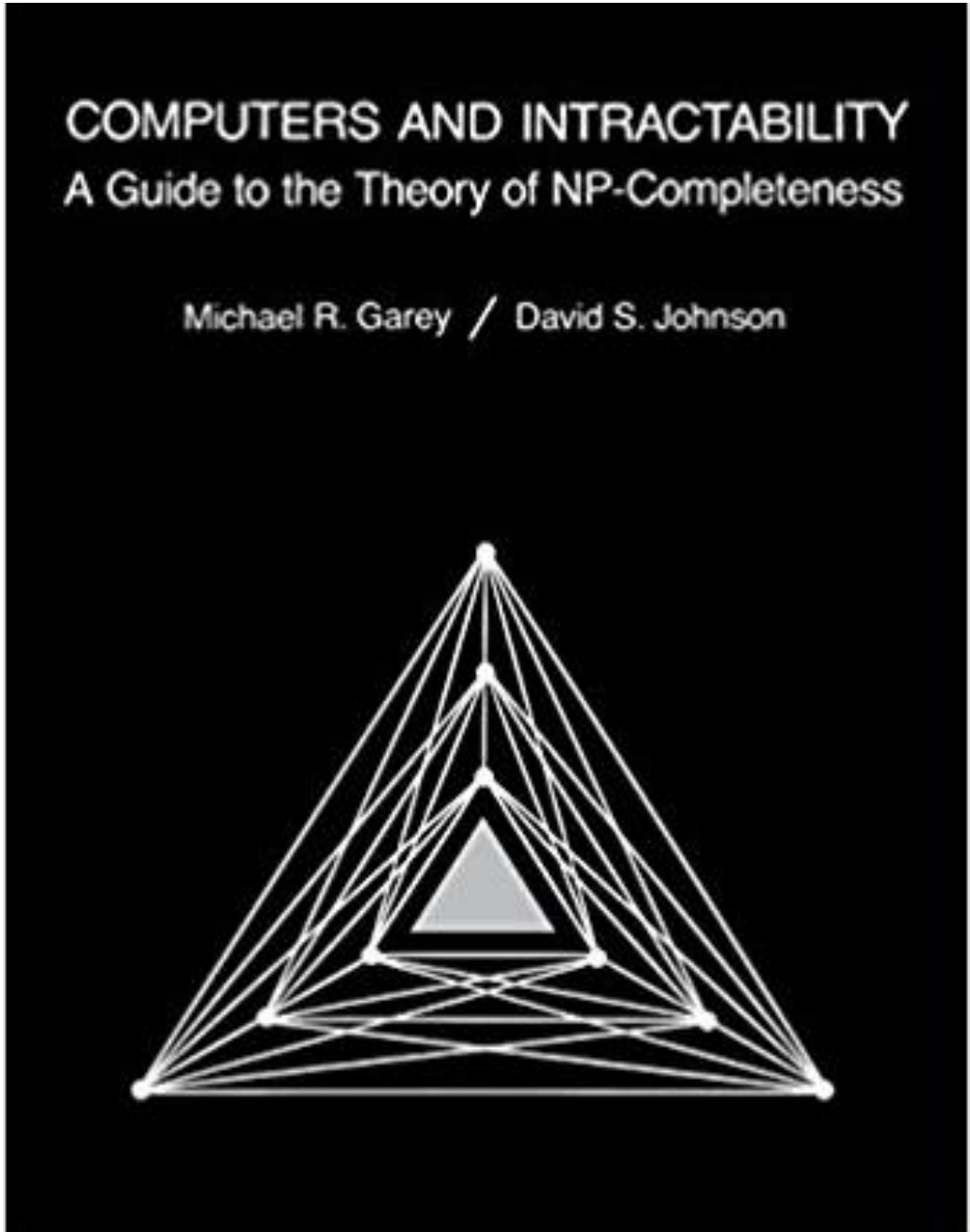
But Wait! There's more!

By 1979, at least 300 problems had been proven NP-complete.

Garey and Johnson put a list of all the NP-complete problems they could find in this textbook.

Took them almost 100 pages to just list them all.

No one has made a comprehensive list since.



NP-Complete Problems

But Wait! There's more!

In the last month, mathematicians and computer scientists have put papers on the arXiv claiming to show (at least) 10 more problems are NP-complete.

If you spend enough time trying to use computers to solve your problems, you will run into an NP-complete problem sooner or later.

What do you do?

Dealing with NP-Completeness

Option 1: Maybe it's a special case we understand

Maybe you don't need to solve the general problem, just a special case

- 2-COLOR vs. 3-COLOR

Option 2: Maybe it's a special case we *don't* understand (yet)

There are algorithms that are known to run quickly on "nice" instances. Maybe your problem has one of those.

One approach: Turn your problem into a SAT instance, find a solver and cross your fingers.

Dealing with NP-Completeness

Option 3: Approximation Algorithms

You might not be able to get an exact answer, but you might be able to get close.

Optimization version of Traveling Salesperson
Given a weighted graph, find a tour (a walk that visits every vertex and returns to its start) of minimum weight.

Algorithm:
Find a minimum spanning tree.
Have the tour follow the visitation order of a DFS of the spanning tree.
Theorem: This tour is at most twice as long as the best one.

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

Why should you care about this problem?

It's your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining problems they listed)

To get a Turing Award

Why should you care about P vs. NP

Most computer scientists are convinced that P≠NP.

Why should you care about this problem?

It's your chance for:

$1,000,000. The Clay Mathematics Institute will give $1,000,000 to whoever solves P vs. NP (or any of the 5 remaining problems they listed)

To get ~~a Turing Award~~ the Turing Award renamed after you.

Why Should You Care if P=NP?

Suppose P=NP.

Specifically that we found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

- $1,000,000 from the Clay Math Institute obviously, but what's next?

Why Should You Care if P=NP?

We found a genuinely in-practice efficient algorithm for an NP-complete problem. What would you do?

- Another $5,000,000 from the Clay Math Institute
- Put mathematicians out of work.
- Decrypt (essentially) all current internet communication.
- A world where P=NP is a very very different place from the world we live in now.

Why Should You Care if P≠NP?

We already expect P≠NP. Why should you care when we finally prove it?

P≠NP says something fundamental about the universe.

For some questions there is not a clever way to find the right answer
- Even though you'll know it when you see it.

There is actually a way to obscure information.

Why Should You Care if P≠NP?

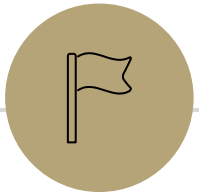
To prove P≠NP we need to better understand the differences between problems.

- Why do some problems allow easy solutions and others don't?
- What is the structure of these problems?

We don't care about P vs NP just because it has a huge effect about what the world looks like.

We will learn a lot about computation along the way.

This is a good time for questions



Optional: Graph practice

Designing New Algorithms

When you need to design a new algorithm on graphs, whatever you do is probably going to take at least $\Omega(m + n)$ time.

So you can run any $O(m + n)$ algorithm as "preprocessing"

Finding connected components (undirected graphs)

Finding SCCs (directed graphs)

Do a topological sort (DAGs)

Designing New Algorithms

Finding SCCs and topological sort go well together:

From a graph G you can define the "meta-graph" G^{SCC}
(aka "condensation", aka "graph of SCCs")

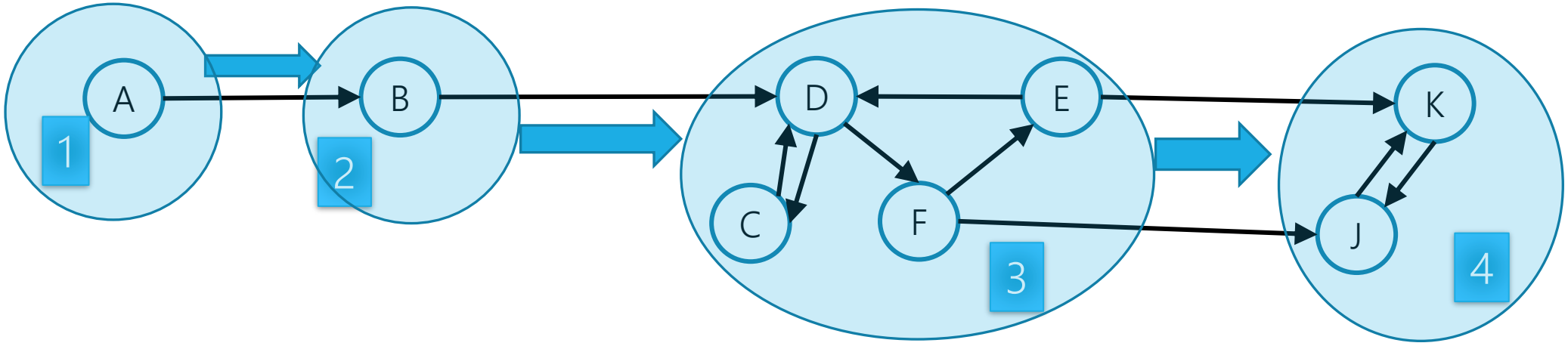
G^{SCC} has a vertex for every SCC of G

There's an edge from u to v in G^{SCC} if and only if there's an edge in G from a vertex in u to a vertex in v.

Why Find SCCs?

Let's build a new graph out of them! Call it G^{SCC}

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.



Designing New Graph Algorithms

Not a common task – most graph problems have been asked before.

When you need to do it, Robbie recommends:

Start with a simpler case (topo-sorted DAG, or [strongly] connected graph).

A common pattern:

1. Figuring out what you'd do if the graph is strongly connected

2. Figuring out what you'd do if the graph is a topologically ordered DAG

3. Stitching together those two ideas (using G^{SCC}).

Graph Modeling

But…Most of the time you don't need a new graph algorithm.

What you need is to figure out what graph to make and which graph algorithm to run.

"Graph modeling"

Going from word problem to graph algorithm.

Often finding a clever way to turn your requirements into graph features.

Mix of "standard bag of tricks" and new creativity.

Graph Modeling Process

1. What are your fundamental objects?
   - Those will probably become your vertices.
2. How are those objects related?
   - Represent those relationships with edges.
3. How is what I'm looking for encoded in the graph?
   - Do I need a path from s to t? The shortest path from s to t? A minimum spanning tree? Something else?
4. Do I know how to find what I'm looking for?
   - Then run that algorithm/combination of algorithms
   - Otherwise go back to step 1 and try again.

Scenario #1

You've made a new social networking app, Convrs. Users on Convrs can have "asymmetric" following (I can follow you, without you following me). You decide to allow people to form multi-user direct messages, but only if people are probably in similar social circles (to avoid spamming).

You'll allow a messaging channel to form only if for every pair of users a,b in the channel: a must follow b or follow someone who follows b or follow someone who follows someone who follows b, or …
And the same for b to a.

You'd like to be able to quickly check for any new proposed channel whether it meets this condition.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #1

You've made a new social networking app, Convrs. Users on Convrs can have "asymmetric" following (I can follow you, without you following me). You decide to allow people to form multi-user direct messages, but only if people are probably in similar social circles (to avoid spamming).

You'll allow a messaging channel to form only if for every pair of users a,b in the channel: a must follow b or follow someone who follows b or follow someone who follows someone who follows b, or …
And the same for b to a.

You'd like to be able to quickly check for any new proposed channel whether it meets this condition.

What are the vertices?

Users

What are the edges?

Directed – from u to v if u follows v

What are we looking for?

If everyone in the channel is in the same SCC.

What do we run?

Find SCCs, to test a new channel, make sure all are in same component.

Scenario #2

Sports fans often use the “transitive law” to predict sports outcomes.
In general, if you think A is better than B, and B is also better than C, then you expect that A is better than C.

Teams don’t all play each other – from data of games that have been played, determine if the “transitive law” is realistic, or misleading about at least one outcome.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #2

Sports fans often use the "transitive law" to predict sports outcomes -- .
In general, if you think A is better than B, and B is also better than C, then you expect that A is better than C.

Teams don't all play each other – from data of games that have been played, determine if the "transitive law" is realistic, or misleading about at least one outcome.

What are the vertices?

Teams

What are the edges?

Directed – Edge from u to v if u beat v.

What are we looking for?

A cycle would say it's not realistic.
OR a topological sort would say it is.

What do we run?

Cycle-detection DFS.
a topological sort algorithm (with error detection)

Scenario #3

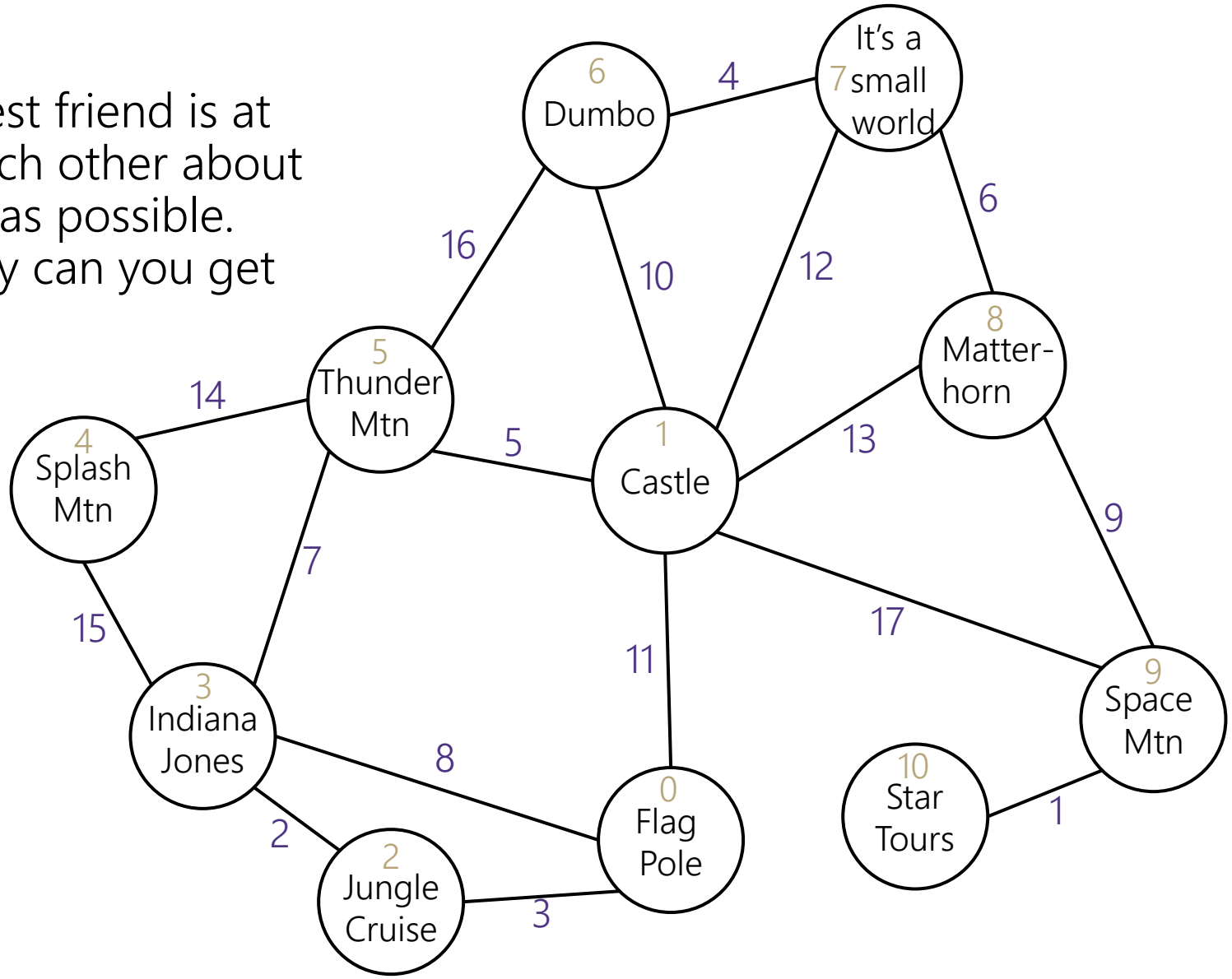
You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

What are the edges?

What are we looking for?

What do we run?



Scenario #3

You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

Rides

What are the edges?

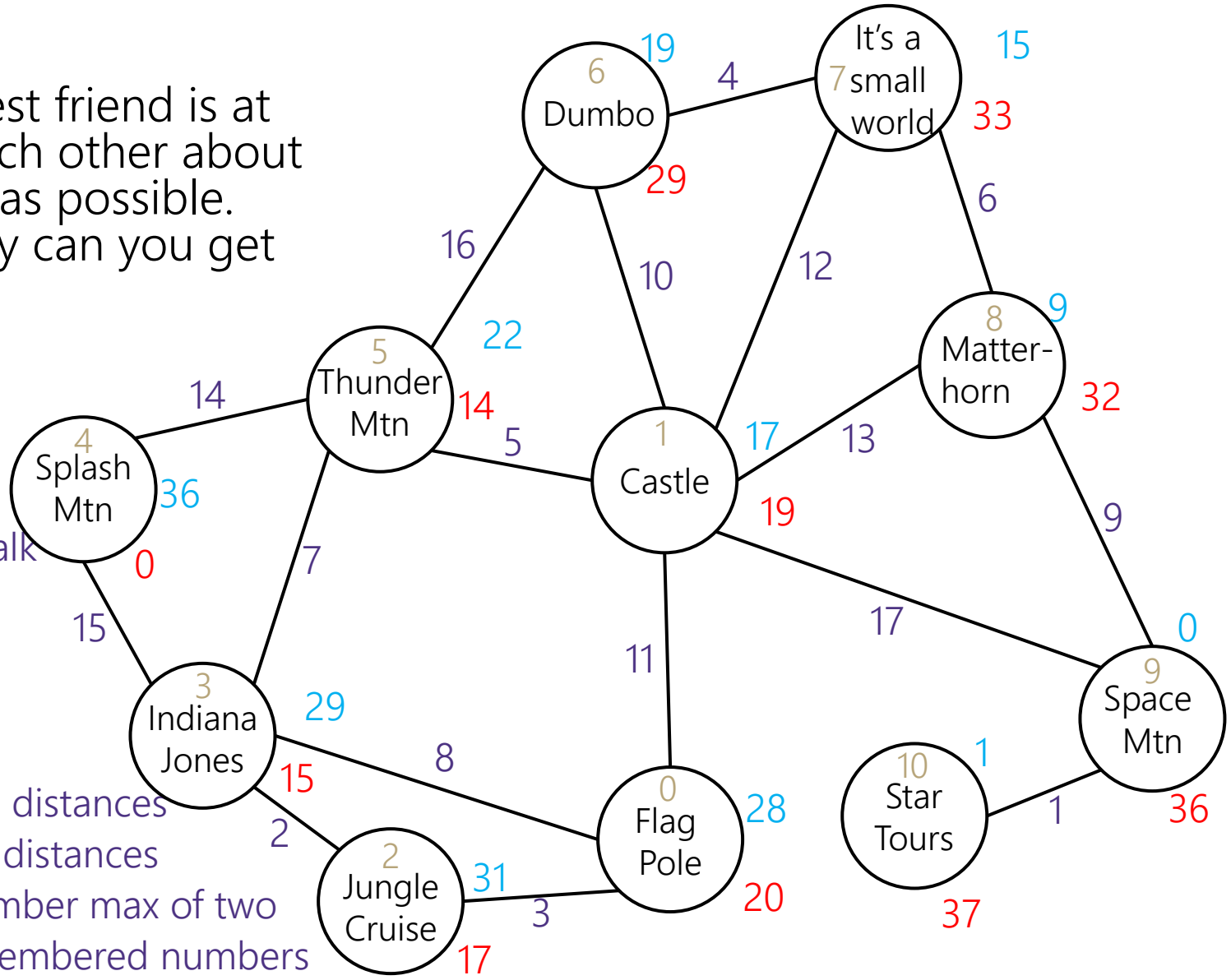
Walkways with how long it would take to walk

What are we looking for?

- The "midpoint"

What do we run?

- Run Dijkstra's from Splash Mountain, store distances
- Run Dijkstra's from Space Mountain, store distances
- Iterate over vertices, for each vertex remember max of two
- Iterate over vertices, find minimum of remembered numbers



Scenario #4

You're a Disneyland employee, working the front of the Splash Mountain line. Suddenly, the crowd-control gates fall over and the line degrades into an unordered mass of people.

Sometimes you can tell who was in line before who; for other groups you aren't quite sure. You need to restore the line, while ensuring if you **knew** A came before B before the incident, they will still be in the right order afterward.

What are the vertices?

People

What are the edges?

Edges are directed, have an edge from X to Y if you know X came before Y.

What are we looking for?

- A total ordering consistent with all the ordering we do know.

What do we run?

- Topological Sort!