Reductions And More on Graphs

CSE 332 Spring 25
Lecture 18

Running Time Analysis

```
Dijkstra(Graph G, Vertex source)
    initialize distances to ∞, source.dist to 0
        mark all vertices unprocessed
        initialize MPQ as a Min Priority Queue
        add source at priority 0
        while(MPQ is not empty){
                u = MPQ.removeMin()
                foreach(edge (u,v) leaving u){
                        if(u.dist+weight(u,v) < v.dist){
                                if(v.dist == ∞ ) //if v not in MPQ
                                        MPQ.insert(v, u.dist+weight(u,v))
                                else
                                        MPQ.decreaseKey(v, u.dist+weight(u,v))
                                v.dist = u.dist+weight(u,v)
                                v.predecessor = u
            }
                }
        mark u as processed
        }
```

With standard heaps:

$$\Theta(n \log n + m \log n)$$

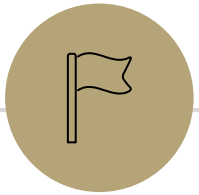
With Fibonacci heaps:

decreasePriority is (amortized) $O(1)$

so total:

$$\Theta(n \log n + m)$$

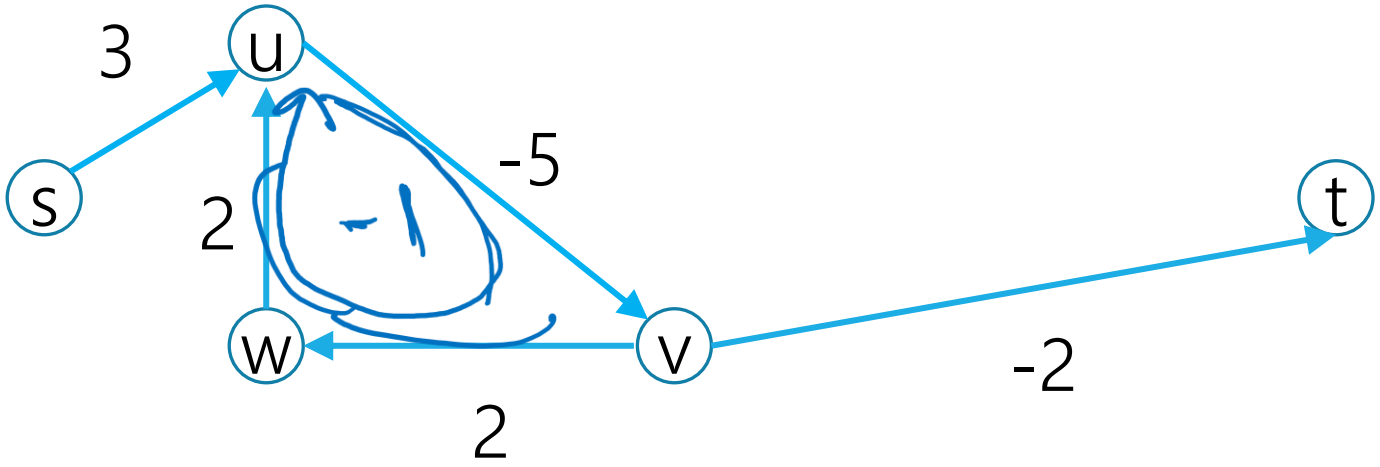
We'll clarify which version we're looking for.



Negative Edge Weights

Negative Edge Weights

What's the shortest way to get from s to t?



s, u,v,w, u,v,w, u,v,w, …

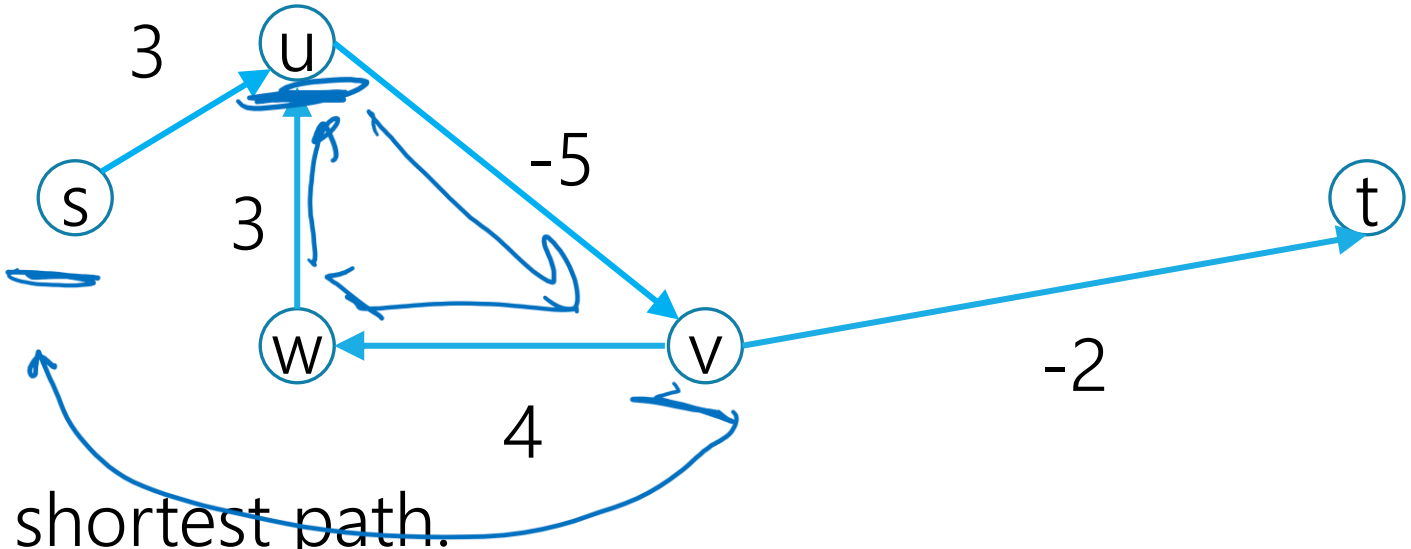
There is no shortest way. You can always go around u,v,w once more.

If there's a **negative weight cycle** shortest paths are **undefined.**

Undefined means "there is no correct answer" (or "$-\infty$ is the closest thing to a correct answer")

Negative Edge Weights

What's the shortest way to get from s to t?



s,u,v,t is the shortest path.

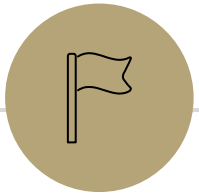
There is a correct answer, but Dijkstra's might not find it!

Negative Edge Weights

If there are negative edge weights, but no negative weight cycle, shortest paths are still defined.

Dijkstra's is only guaranteed to work when edge weights are positive

- For GoogleMaps positive edge weights definitely make sense.
- Sometimes negative weights make sense too.
- **Dijkstra's algorithm doesn't work for those graphs**
- There are other algorithms that do work (ask Robbie later)



Reductions

Designing a new algorithm

Sometimes, the way to design an algorithm is:

- Find something that works in a special case
- Figure out why it works there
- Adapt it to work in the general case.

That's what we did for Dijkstra's:

- Start with BFS (worked in special case---unweighted graphs)
- Realized it worked by processing in distance order
- Adapt to process in distance order with weighted edges.

Sometimes we want a different process…

Reductions

Another way to design an algorithm is via a "reduction"

- Figure out [or ask someone else] how to solve a related problem (call that "problem B")
- Modify your *input,* then call the library for problem B
- Use the answer for problem B, to answer your problem ("problem A")

This design technique is called a "reduction"

We say that "problem A reduces to problem B"

Weighted Graphs: Take 2

Reduction (informally)

Using an algorithm for Problem B to solve Problem A.

You already do this all the time.

Any time you use a library, you're reducing your problem to the one the library solves.

Can we reduce finding shortest paths on weighted graphs to finding them on unweighted graphs?

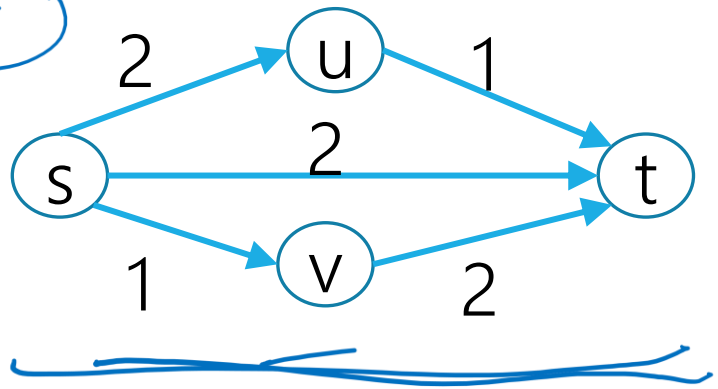
Weighted Graphs Take 2

Can we reduce “Shortest Paths on a weighted graph” to “shortest paths on an unweighted graph”?

I.e., someone wrote you a library function `UnweightedSP` can you use that to find the shortest paths in a weighted graph?

`UnweightedSP` probably does BFS…but we’re not going to care exactly how it does what it does (just that it does it correctly).

Weighted Graphs: A Reduction



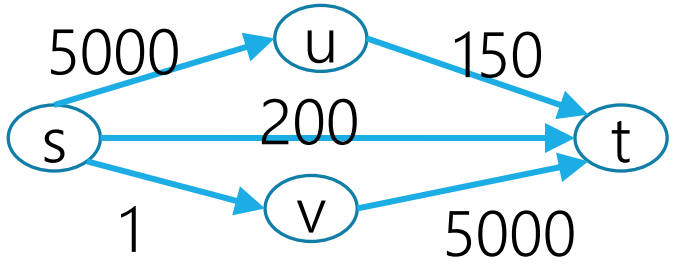
Transform Input

Unweighted Shortest Paths

Transform Output

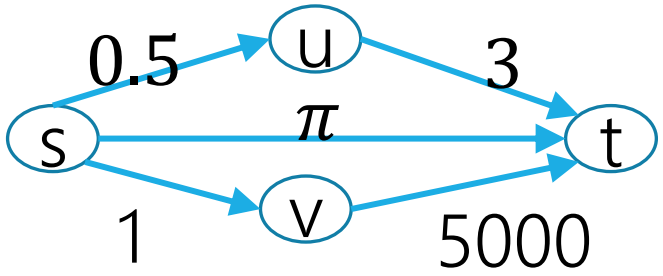
Weighted Graphs: A Reduction

What is the running time of our reduction on this graph?



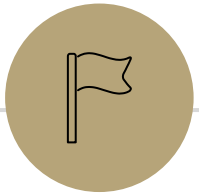
O(|V|+|E|) of the **modified** graph, which is…slow.

Does our reduction even work on this graph?



Ummm….

Tl;dr: If your graph's weights are all small positive integers, this reduction might work great.
Otherwise we would use Dijkstra's



Coloring

Reduce 3-coloring to 4-coloring

Let's reduce 3-coloring to 4-coloring

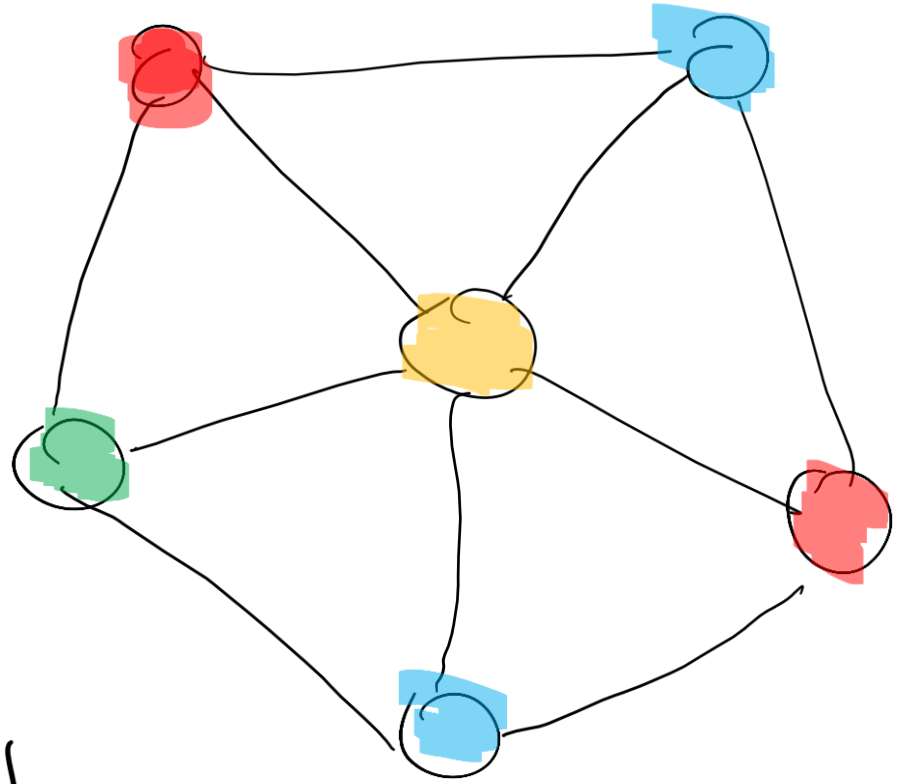
3-coloring

Input: Undirected Graph G
Output: `True` if the vertices of G can be labeled with red,green, and blue so that no edge has both of its endpoints colored the same color. `False` if it cannot.

4-coloring

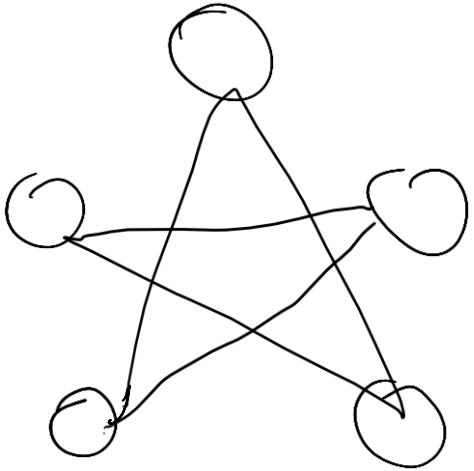
Input: Undirected Graph G
Output: `True` if the vertices of G can be labeled with red,green, blue, and orange so that no edge has both of its endpoints colored the same color. `False` if it cannot.

Are these 3-colorable? 4-colorable?

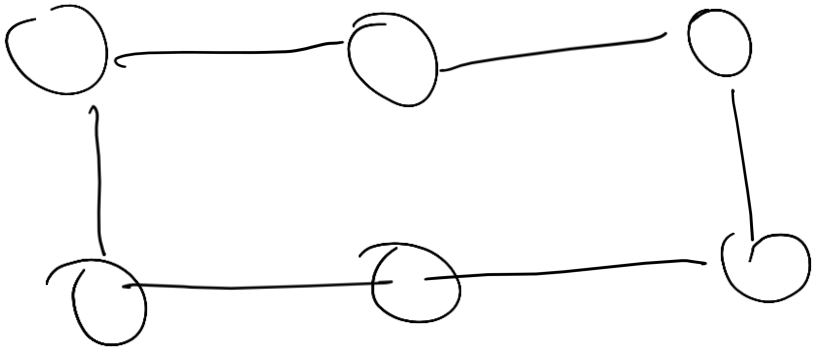


1

2



3



Reduce 3-coloring to 4-coloring

Given a graph G, figure out whether it can be 3-colored, by using an algorithm that figures out whether it can be 4-colored.

Usual outline:

Transform G into an input for the 4-coloring algorithm

Run the 4-coloring algorithm

Transform the answer from the 4-coloring algorithm into the answer for G for 3-coloring

Reduction

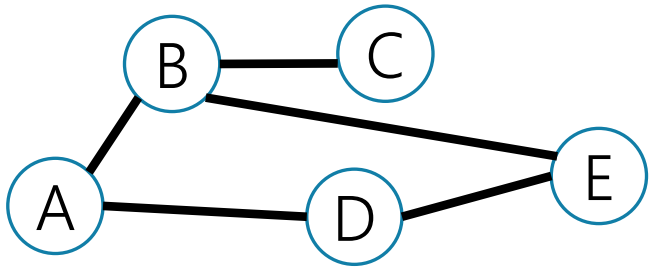
If we just ask the 4-coloring algorithm about G, it might use 4 colors…we can't get it to use just 3…

…unless…

Unless we force it not to, by adding extra vertices that **force** the 3-coloring algorithm to "use up" one color on the extra vertices, leaving only two colors for the "real" vertices.

Reduction

```
3ColorCheck(Graph G)
    Let H be a copy of G
    Add a vertex to H, attach it to all other
vertices.
    Bool answer = 4ColorCheck(H)
    return answer //don't need any modification!
```



Transform Input

4ColorCheck algorithm

Transform Output

Correctness?

```
3ColorCheck(Graph G)
     Let H be a copy of G
     Add a vertex to H, attach it to all
       other vertices.
     Bool answer = 4ColorCheck(H)
     return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

If the correct answer for G is NO, then we say NO

Correctness?

```
2ColorCheck(Graph G)
     Let H be a copy of G
     Add a vertex to H, attach it to all
       other vertices.
     Bool answer = 3ColorCheck(H)
     return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

If G is 3-colorable, then H will be 4-colorable – you can extend a 3-color labeling of G to 4 colors on H by making the new vertex the new color. All the edges in G have different colors (because we started with a 3-coloring) and any added edge has different endpoints (because v is a new color) so 4ColorCheck returns True and we return True!

If the correct answer for G is NO, then we say NO

Correctness?

```
2ColorCheck(Graph G)
     Let H be a copy of G
     Add a vertex to H, attach it to all
       other vertices.
     Bool answer = 3ColorCheck(H)
     return answer
```

TWO statements to prove: (“two directions”)

If the correct answer for G is YES, then we say YES

The new vertex can be a new color!

If the correct answer for G is NO, then we say NO

So we **can’t 3-color G**. That’s going to be hard to work with.

Take the contrapositive!!

Correctness?

```
2ColorCheck(Graph G)
     Let H be a copy of G
     Add a vertex to H, attach it to all
       other vertices.
     Bool answer = 3ColorCheck(H)
     return answer
```

TWO statements to prove: ("two directions")

If the correct answer for G is YES, then we say YES

The new vertex can be a new color!

If the correct answer for G is NO, then we say NO

We want to show instead: If we say YES, then the correct answer is YES.
If we say YES, then 4ColorCheck(H) must have returned YES, what does a 4-coloring of H look like? The added vertex must be a different color than all the other vertices (otherwise it's not a valid coloring – there's an edge between the added vertex and all others). So deleting the added vertex we get a 3-coloring of G. So the right answer is YES!!

Correctness

Two DIFFERENT statements

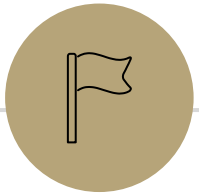
Correct Answer YES → Our algorithm says YES

If G is 3-colorable, then H will be 4-colorable – you can extend a 3-color labeling of G to 4 colors on H by making the new vertex the new color. All the edges in G have different colors (because we started with a 3-coloring) and any added edge has different endpoints (because v is a new color) so 4ColorCheck returns True and we return True!

Our algorithm says YES → Correct Answer YES

We want to show instead: If we say YES, then the correct answer is YES.

If we say YES, then 4ColorCheck(H) must have returned YES, what does a 4-coloring of H look like? The added vertex must be a different color than all the other vertices (otherwise it's not a valid coloring – there's an edge between the added vertex and all others). So deleting the added vertex we get a 3-coloring of G. So the right answer is YES!!



Why Care about reductions?

Why Care About reductions

A reduces to B says "If you can solve problem B then you can solve problem A"

- Saves code-writing time (or algorithm designing time!)
- Once someone wrote the algorithm for B, the algorithm for A is easy to write.

But take a contrapositive

A reduces to B also says "If you cannot solve problem A, then you cannot solve problem B.

- If you instead know (or believe) that A is difficult, you can convince yourself that B is difficult as well.

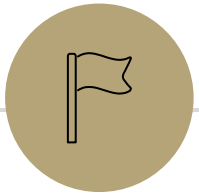
Gracefully Giving Up

Reductions give computer scientists a graceful way to stop trying to find a better algorithm.

If you were trying to design an $O(n \log \log n)$ algorithm for sorting, the proof we did last week says you can officially give up.

You can bootstrap that into arguments that algorithms for other problems aren't possible; you'll do this (among other things) on the next exercise.

Reductions are also the core of "NP-completeness" as a concept (and indeed most of complexity theory) they'll come back in a few weeks.



More Graph Applications

Another Application of Shortest Paths

Shortest path algorithms are obviously useful for GoogleMaps.

The wonderful thing about graphs is they can encode **arbitrary** relationships among objects.

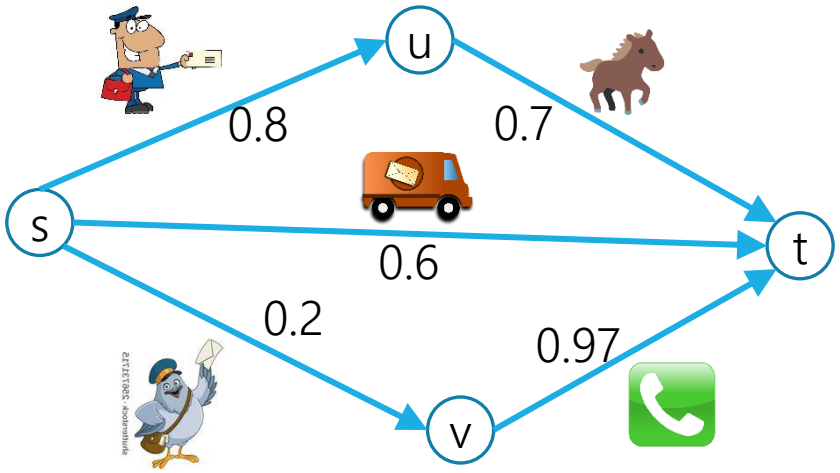
Details here aren't the main thing…

I want you to see that these algorithms have non-obvious applications.

I want you to do a reduction.

Another Application of Shortest Paths

I have a message I need to get from point s to point t. But the connections are unreliable. What path should I send the message along so it has the best chance of arriving?



Maximum Probability Path

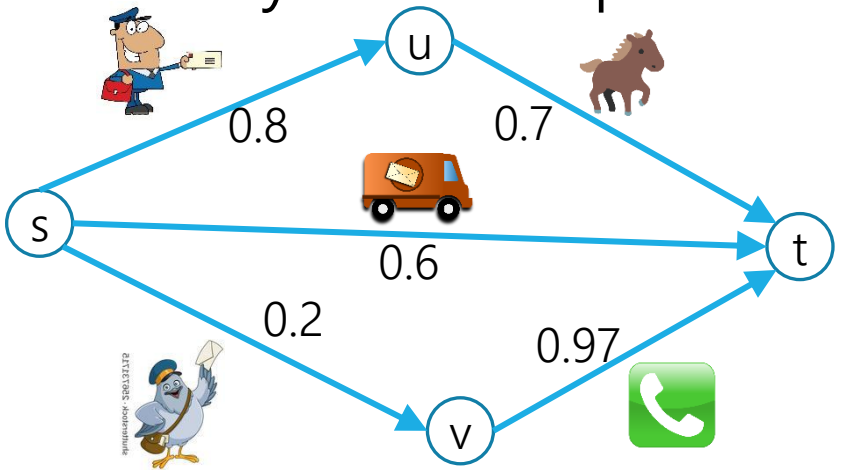
Given: a directed graph G, where each edge weight is the probability of successfully transmitting a message across that edge
Find: the path from s to t with maximum probability of message transmission

Another Application of Shortest Paths

Let each edge's weight be the probability a message is sent successfully across the edge.

What's the probability we get our message all the way across a path?

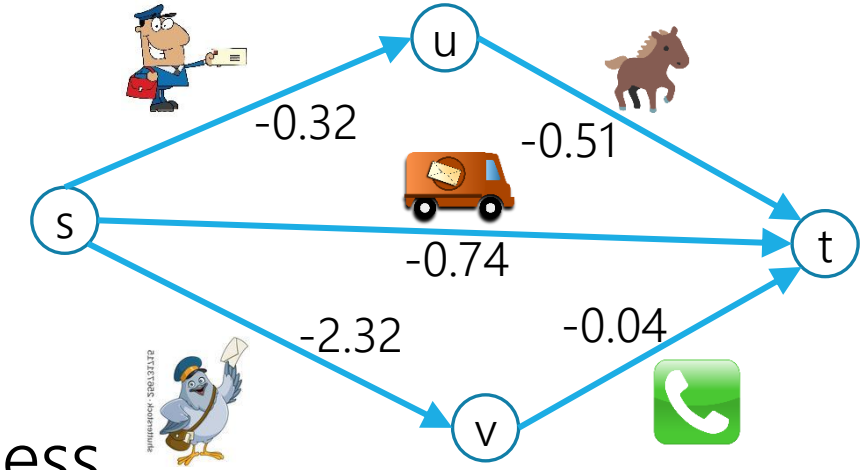
- It's the product of the edge weights.



We only know how to handle sums of edge weights.

Is there a way to turn products into sums?

Another Application of Shortest Paths



We've still got two problems.

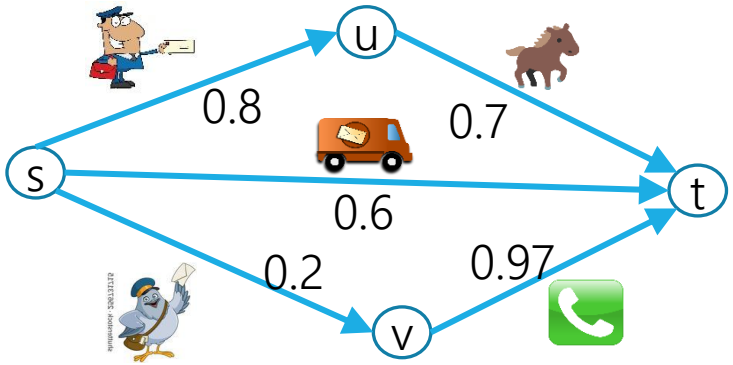
1. When we take logs, our edge weights become negative.

2. We want the *maximum* probability of success, but that's the longest path not the shortest one.

Multiplying all edge weights by negative one fixes both problems at once!

We **reduced** the maximum probability path problem to a shortest path problem by taking $-\log()$ of each edge weight.

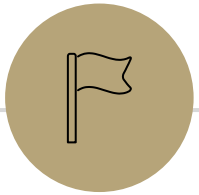
Maximum Probability Path Reduction



Transform Input

Weighted Shortest Paths

Transform Output



Graph Modeling

Graph Modeling

Much 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 #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?

You can modify DFS to find cycles (ask Robbie later).
a topological sort algorithm (with error detection)

Scenario #3

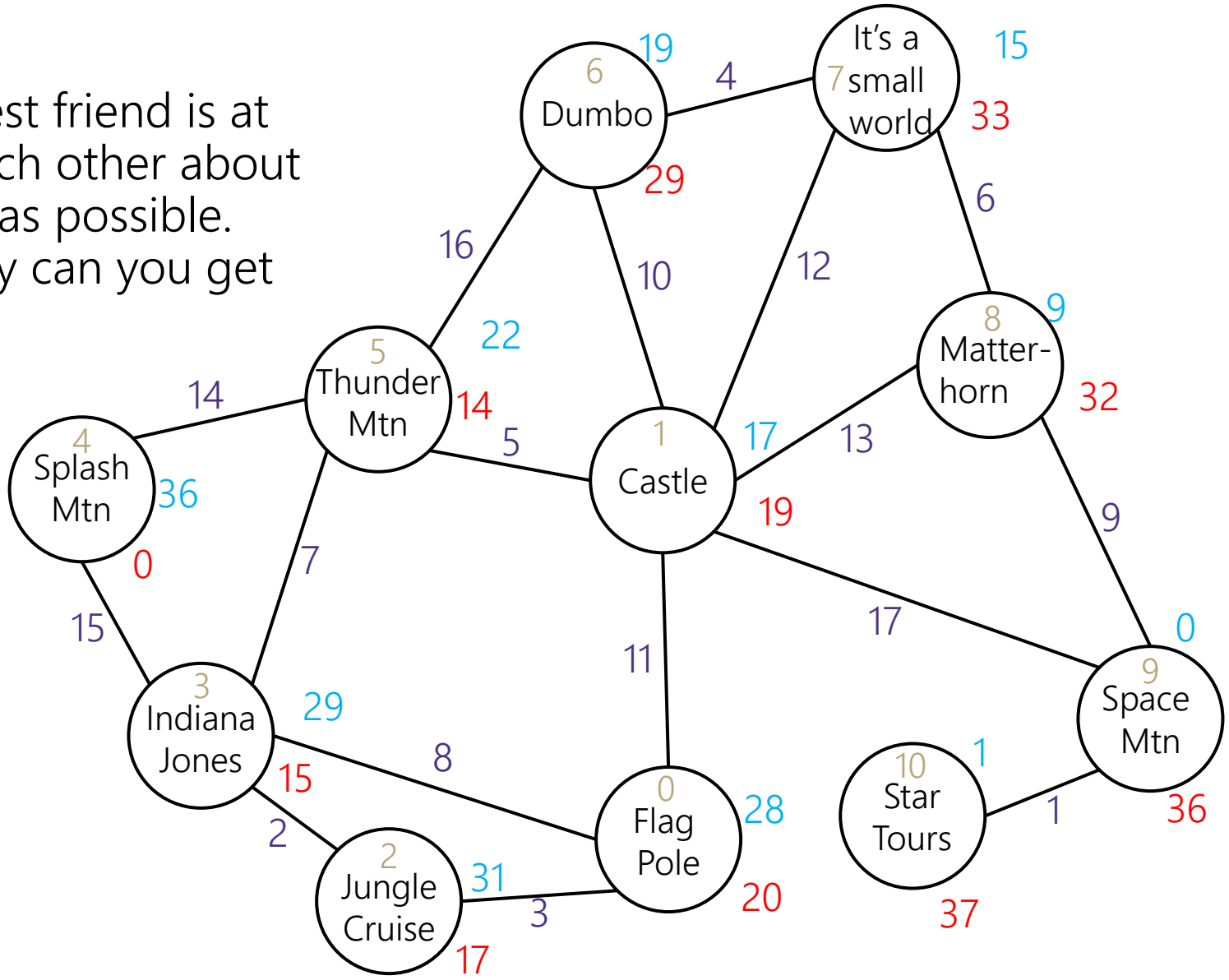
You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

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?

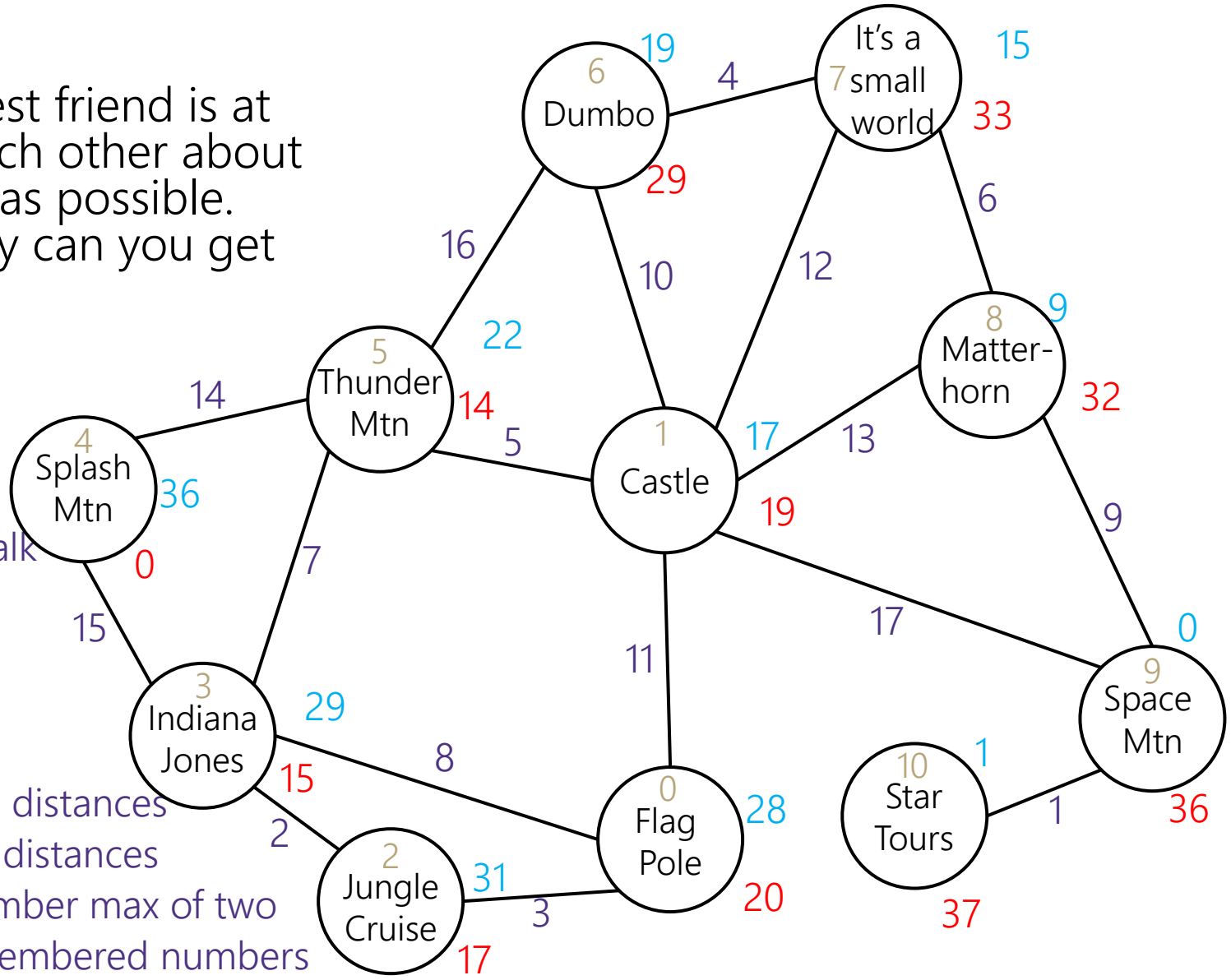
Walkways with how long it would take to walk

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?

What are the edges?

What are we looking for?

What do we run?

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!