Graphs

Gabe Johnson

Applied Data Structures & Algorithms

University of Colorado-Boulder

This sequence introduces a new category of abstract data type called Graphs. Graphs are fun. They let you model the relationship between things in many ways that you can't really do with trees. They show up everywhere, from social networks to robot motion planning algorithms.

A generic graph can be really useful, in the same way that a generic tree is. But you can also specialize the idea of a Graph by imposing more rules on it, sort of like you can specialize trees into binary search trees or B-trees.

We're going to spend this sequence talking about graphs in a generic way, though I'll mention specialized graphs a few times, just to keep things dramatic.

And then in the next sequence we'll delve into a specific kind of graph called a state machine.

Episode 1

Introduction to Graphs

Revisit Lists & Trees

Traverse from beginning to end

```
// linked list traversal
while (cursor->next != NULL) {
  visit(cursor);
  cursor = cursor->next;
}
```

Traverse with pre-, in-, or post-order

```
// binary tree inorder traversal
void inorder(bt_node* node) {
  if (node != NULL) {
    inorder(node->left);
    visit(node);
    inorder(node->right);
  }
}
```

I'm going to start out by revisiting the two structures from the beginning of the course to give some context for talking about graphs.

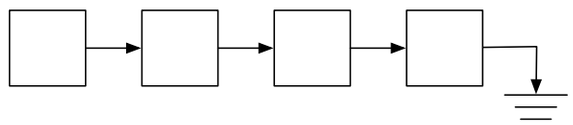
There are algorithms for linked lists and binary trees that you have hopefully become familiar with. If I had to characterize them in terms of how you traverse them, with a linked list you might use a simple loop and a cursor to traverse it linearly from beginning to end.

<next build>

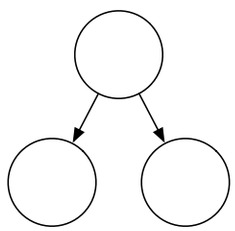
And with a binary tree, you can traverse it in three main ways (pre-order, in-order, post-order), often with recursion.

The exact details depend on what you're doing: looking for a particular value, counting nodes, looking for the end of the list, or a leaf node, and so on.

List & Tree Structure



List: single link

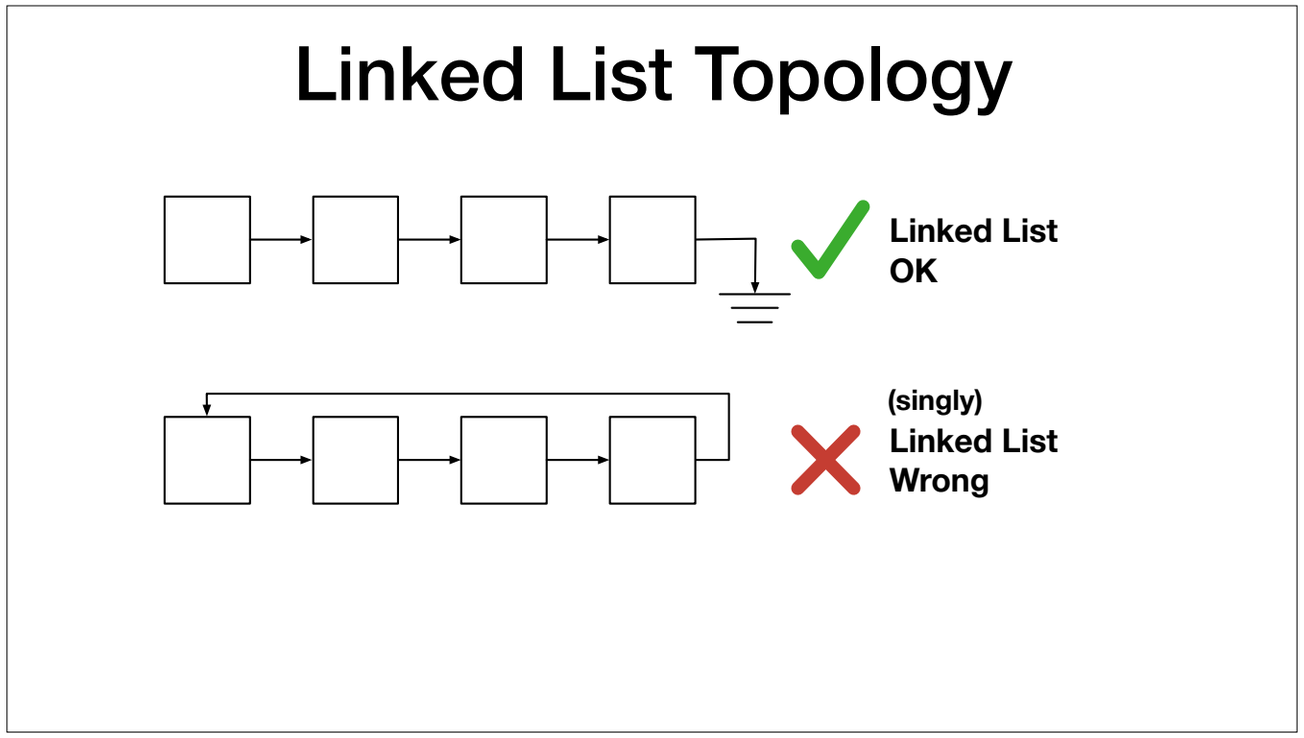


Binary Tree: two links

A Linked List is a simple structure. It is built from nodes that have data and a single link that points at another node.

A binary tree is a little more complicated. A binary tree node contains data and _two_ links: a left child and a right child.

You can think of both linked lists and binary trees as very simple graphs that have rules about how the nodes can be related.

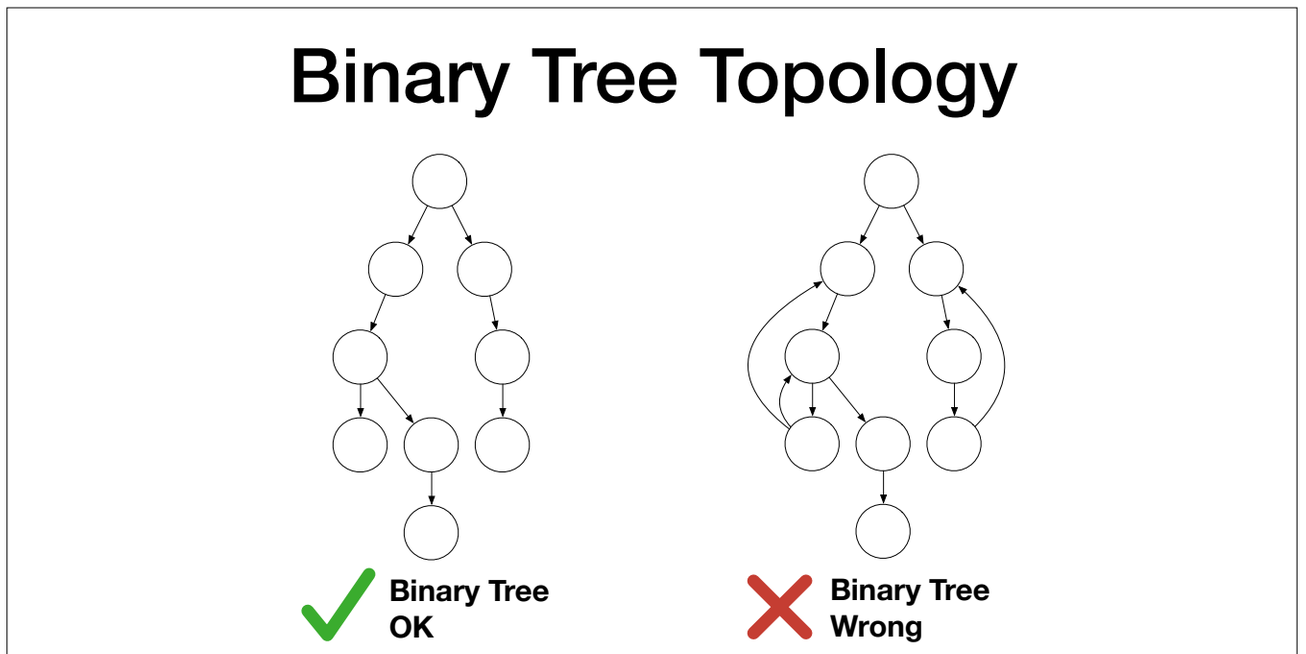


A Graph is a structure that contains nodes, where data lives, and edges, which form relationships between nodes, and sometimes has data attached as well.

We can impose constraints on how we form edges, like we do with Linked Lists and Binary Trees. Here's a correctly formed linked list from how we did it earlier in the homework.

Linked lists edges are directed, and each node only connects to one other.

We impose rules on the _topology_: what can legally be reached by following nodes. So for example, you can't backtrack like this in a linked list **<next>** because the last node should point to NULL. Otherwise we can't traverse it as expected. There are other kinds of structures where this is OK, just not for our Linked List.

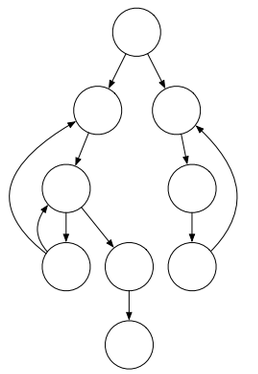


With a binary tree, the nodes can have zero, one, or two children.

<next>

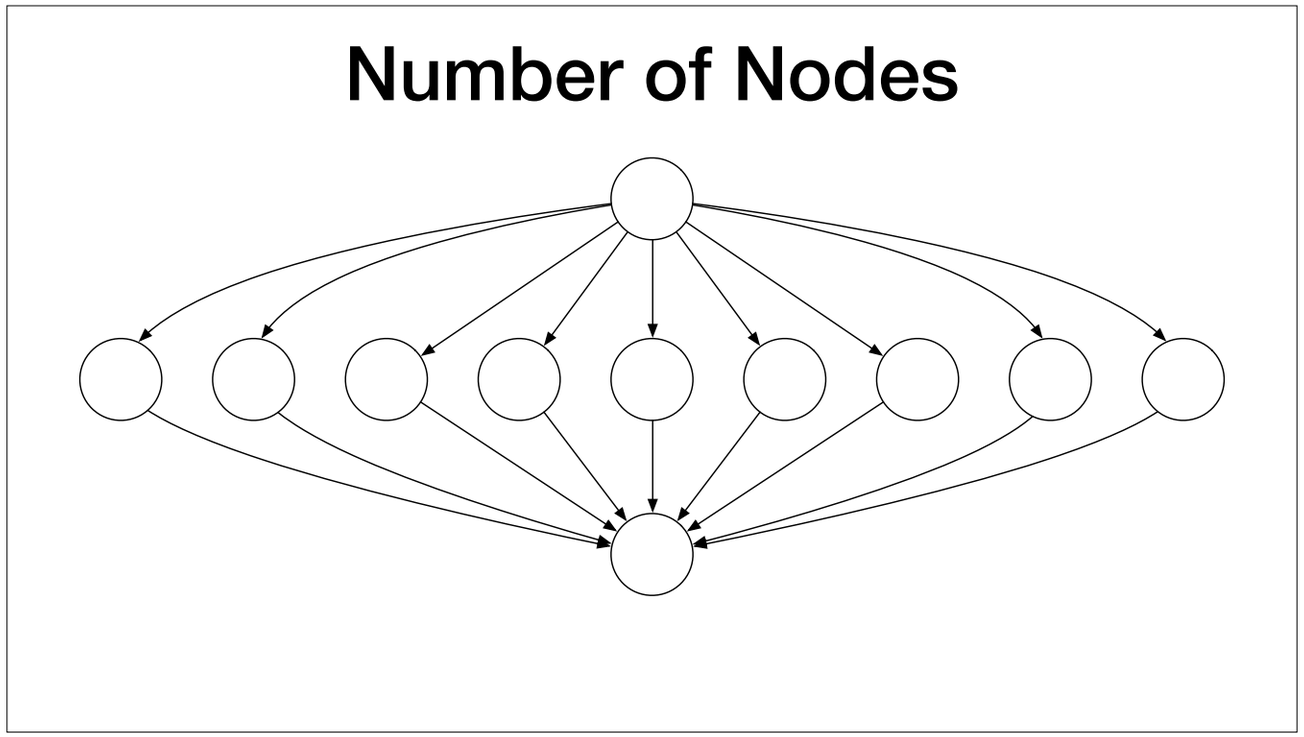
We've also had the implicit rule that a parent or any other ancestor can't be a child's child. Because that would be weird!

Graph Topology



A generic graph, one with the fewest constraints on its topology, just just nodes and edges. So that 'wrong' binary tree from the last slide is a perfectly OK generic graph.

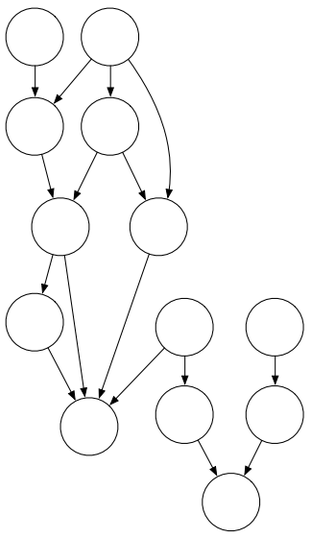
Now to tie this back to the linked list and binary tree topologies. Earlier we had straightforward ways to traverse those structures. But traversing a graph is a little trickier, and explaining that will be the topic of future episodes.



There's no rule about the number of edges a graph node can have entering or leaving it.

This is a perfectly legitimate graph. From now on, assume that the number of edges is unlimited for all graph types unless otherwise noted.

Specialized Graphs



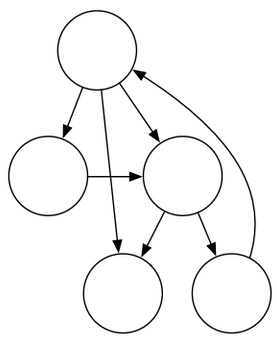
Directed Acyclic Graph

aka DAG

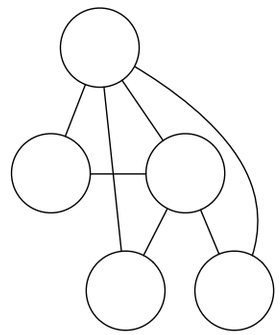
There are lots of kinds of graphs that have some added constraints that give them special powers.

This one is called a Directed Acyclic Graph. It is common enough that it gets its own acronym: DAG. It's a loaded phrase, so let me break that down for you, because it helps introduce a couple of the important concepts with graphs and graph algorithms.

Directed vs Undirected



Directed Graph
“digraph”



Undirected Graph

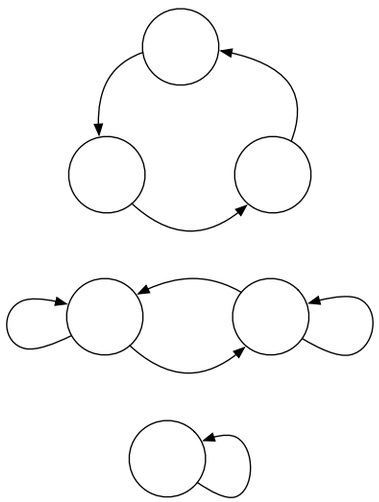
DAGs are Directed, in that the edges have distinct start and end nodes. The direction is indicated with arrowheads.

The opposite of a directed edge is an undirected one. Undirected edges just means that there's a relationship between the nodes, but it doesn't necessarily have any sort of ordering.

If your graph only allows directed edges, we say it is a directed graph. Sometimes you'll see this shortened to just 'digraph'.

If your graph only allows undirected edges, we say it is an undirected graph. You probably won't hear anybody refer it as an 'ungraph', though there's nothing stopping you from being a trendsetter!

Cycles



If the graph doesn’t allow cycles, it is **Acyclic**.

If a graph contains a cycle, it is **Cyclic**.

(All of these are cyclic)

This is a cycle. DAGs are Acyclic because cycles are not allowed. If your graph has at least one cycle, it is called Cyclic.

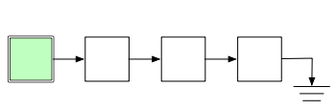
A cycle is a path that takes you from one location and leads you back to that same location. Linked Lists and Binary Trees don't typically allow cycles, and neither do DAGs.

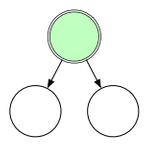
Many graphs exploit cylces. This two-node graph has two nodes but four edges. Notice that the left and right nodes have two edges distinct edges that connect them.

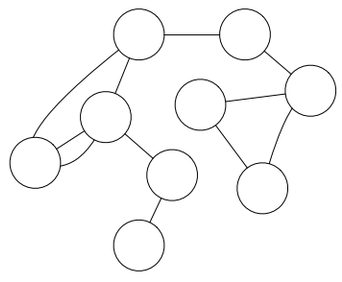
You can even have crazy cycles where an edge starts and ends on the same node.

In that case you can have a one-node graph that also has a cycle. Weirdo graph, but completely legit if your graph type allows it.

No First Node







Where to begin?
Depends on what you're doing.

A graph doesn't necessarily have a 'first node'. We might have reason to choose node over another as a first node based on our use case, or we might look at the structure of the node and determine that one node sticks out being special in some way. A linked list has a start node, **<next>** and a binary tree has, the root, the one at the top.

<next>

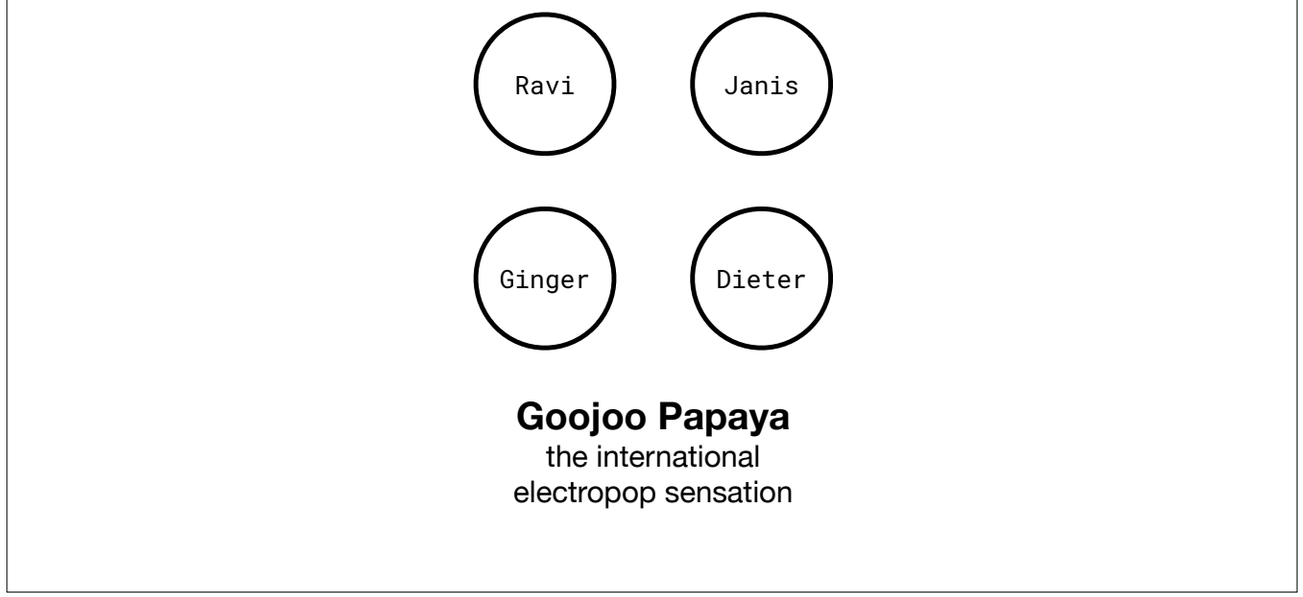
But what about this graph?

Often your use case means one particular node is the right starting place, like if you're measuring the number of hops between a node and every other one. If there is no particular place to start, many algorithms just let you pick one arbitrarily.

Episode 2

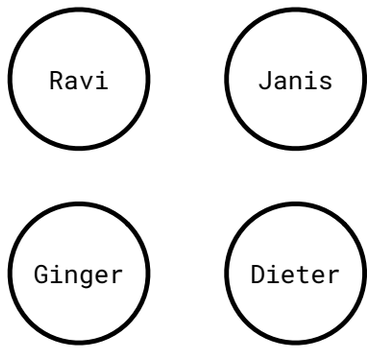
Implementing Graphs

In this episode we're going to cover some of the ways that you can concretely implement graphs. Which approach you take should depend on the use case, though aside from some vague advice, it will be up to you to figure out which is the most appropriate based on what you're doing.



Goojoo Papaya is the international electropop sensation! I'm going to use this made-up band as the basis for the graph examples.

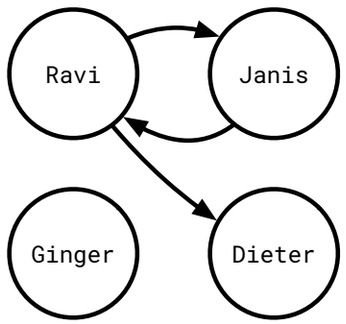
Disconnected Nodes OK



Goojoo Papaya
the international
electropop sensation

And this thing you're looking at is indeed a graph. It is OK for a node, or in this case all nodes, to not have any adjacent edges. It is still a graph.

Adjacency Matrix



A enjoys playing with B

```
int main() {
  adj_node* nodes[4]; // storage for nodes
  bool adj[4][4];     // adjacency matrix
  nodes[0] = mk_adj_node("ravi");
  nodes[1] = mk_adj_node("janis");
  nodes[2] = mk_adj_node("ginger");
  nodes[3] = mk_adj_node("dieter");
  adj[0][1] = true; // ravi to janis
  adj[1][0] = true; // janis to ravi
  adj[0][3] = true; // ravi to dieter
}
```

This graph models which members of Goojoo Papaya enjoy playing with who.

The first main way to represent a graph is with an adjacency matrix. Use a square matrix, such as a 2D array where both the inner and outer arrays have the same length.

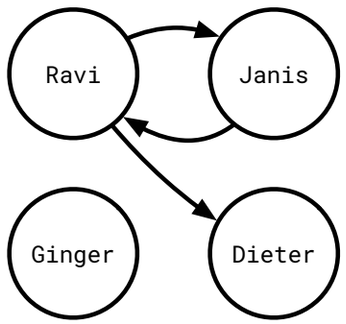
<next>

In this example, we've got four nodes for each member of the Goojoo Papaya: Ravi, Janis, Ginger, and Dieter. So our 2D array is four by four.

To establish a directed relationship between one band member and another, we need to use their index. So since Ravi is at index 0 and Janis at index 1, we set cell 0, 1 to true to note that Ravi likes to work with Janis.

And it is reciprocal, so there's an edge from Janis to Ravi. And while Ravi likes to work with Dieter, Dieter has no particular opinion about Ravi.

Adjacency Matrix



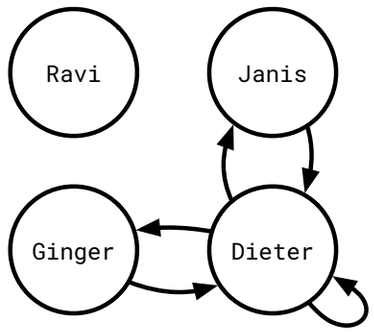
A enjoys playing with B

	Ravi	Janis	Ginger	Dieter
Ravi	0	1	0	1
Janis	1	0	0	0
Ginger	0	0	0	0
Dieter	0	0	0	0

Adjacency Matrix

The directed graph for who likes to play with who looks like this when spelled out as a matrix. Notice that we have 4 nodes, and 16 cells for modeling edges. If you use a 2D array like we're doing here, you'll need N squared edge spots for N nodes.

Adjacency List



A dislikes playing with B

```
struct adj_node {
  string name;
  vector<adj_node*> related;
};

int main() {
  // graph models who dislikes playing with who
  adj_node* ravi = make_node("ravi");
  adj_node* janis = make_node("janis");
  adj_node* ginger = make_node("ginger");
  adj_node* dieter = make_node("dieter");
  janis->related.push_back(dieter);
  ginger->related.push_back(dieter);
  dieter->related.push_back(janis);
  dieter->related.push_back(ginger);
  dieter->related.push_back(dieter);
}
```

Here's a different graph, showing who _doesn't_ like playing with who. Poor Dieter is such a tortured artist, he doesn't like playing with anyone, even himself!

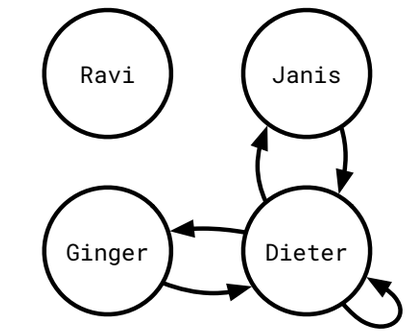
Another way of implementing a graph is to associate each node with some kind of list of related nodes.

<next>

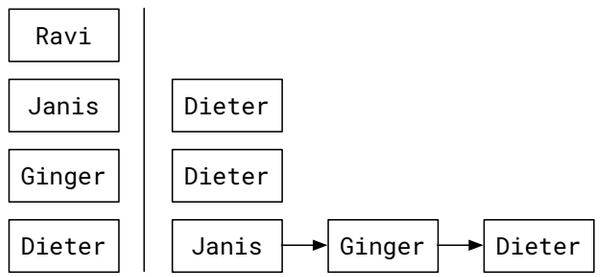
The code here is a non-fancy way of doing that. There's a struct for each node in the graph, and now since we're not referencing them with indices, we can use a pointer to each node instead.

To record the relationship from one node to another, we access each node's list of related nodes, and push the other node onto its list.

Adjacency List



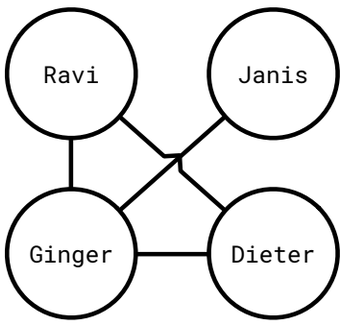
A dislikes playing with B



Adjacency List

Here's a visual for the adjacency list. We have four nodes, and only five edges. There's not a lot of wasted space here. So while we're being more efficient with memory, we're adding some computational complexity because now if we want to look at an edge, we have to search for it. I've drawn the edge lists here as linked lists, so finding a particular edge means potentially scrolling through the entire edge list. But if you use a hash map or something hash-related it will be more efficient.

Undirected



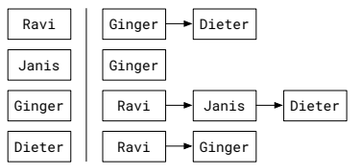
Writes Songs Together

	Ravi	Janis	Ginger	Dieter
Ravi	0	0	1	1
Janis	0	0	1	0
Ginger	1	1	0	1
Dieter	1	0	1	0

Adjacency Matrix

	Ravi	Janis	Ginger	Dieter
Ravi				
Janis	0			
Ginger	1	1		
Dieter	1	0	1	

Triangular Adjacency Matrix



Adjacency List

We've seen directed graphs so far, because we've been modeling one band member's sentiment towards another one. But it is also possible to model an undirected relationship, such as who is writing new material with who.

<next>

To model this with a matrix, we could just use the square array and make sure that whenever we mark an edge from A to B, we also mark it from B to A. That wastes some space because we're storing redundant information.

<next>

One solution to that is to only use a triangular matrix, we're using a lower triangle here.

<next>

But if we're using an adjacency list, we'll end up duplicating the information.

Variations on the Theme

Main Strategies to Implement Graphs:

- Adjacency Matrix
- Adjacency List

Ways to spice it up:

- Weighted Edges
- Edges as compound structs/objects
- Use node pairs as key to identify edges
- Separate edge sets to overlay on nodes

The two main approaches I've shown just now are:

- Adjacency Matrix
- Adjacency List

There are lots of ways you can code up graphs, but all that I'm aware of can broadly be described as one of these.

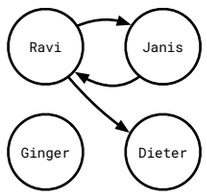
The adjacency matrix shown earlier only recorded a boolean value, true or false, if there was an edge. But you could instead use a number to record a weight, or maybe a string to record the nature of relationship, or have some more complex struct to model the relationship.

And the same is true for the adjacency list approach - you could model edges in many ways.

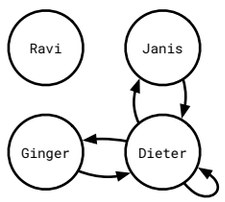
Another approach to record edges is to use some pair of nodes, say Janis and Ginger as a key into a hash map, where the value is the edge, if it exists. This way you keep your nodes and edges separate, which can have advantages.

You could even have different edge sets applied to the same graph!

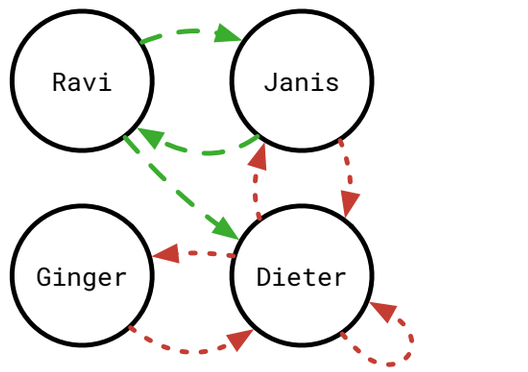
Variations on the Theme



A enjoys playing with B



A dislikes playing with B



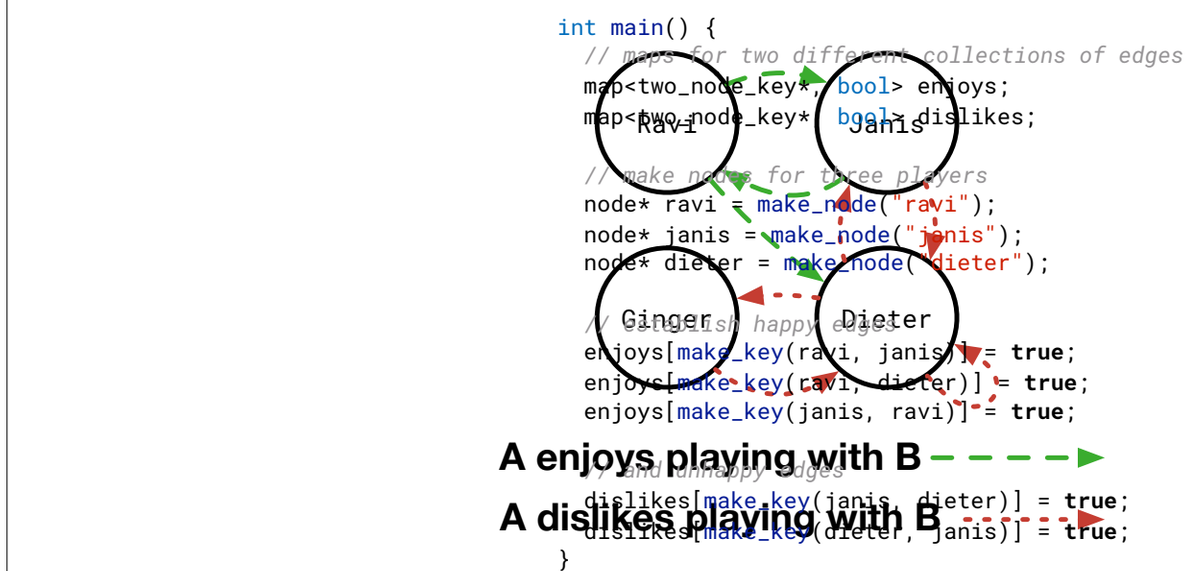
A enjoys playing with B

A dislikes playing with B

For example, we saw the graphs that showed which members of Goojoo Papaya enjoyed playing with the others, and the other one of who _doesn't_ like playing with who. Anyway, we could overlay these two different kinds of edges on the same graph and get something like **<next>** this.

And if you wanted to do that sort of thing, you'd have to implement your graph with that use case in mind. We're not going to so anything fancy like that in this class. But do keep that in mind. I've personally needed to use edge overlays many times.

Variations on the Theme



Here's what I mean by that in code. For each kind of edge, create a separate container. Here I'm using maps, but I could have used a set instead.

For each 'enjoys' relationship, make a key using the source and destination nodes, and add it to the container. Here we're relying on the key to have a good hash function! But you knew that.

And then for each 'dislikes' relationship you do the same thing.

So in this strategy, you're keeping the different kinds of edges completely separated.

Episode 3

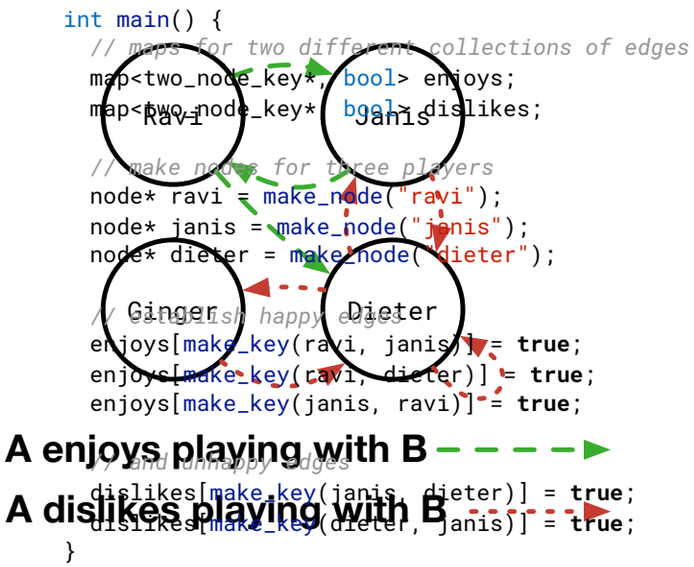
Graph Structure & Metadata

Graphs are one of the most versatile abstract data types. The particular way you wire them up, and what specific data is needed, will almost always be on a case-by-case basis.

Happily, we can vary the way we structure the graph to make it suit our needs. And we can also use metadata - that means 'data about other data'. The metadata can be applied to the whole graph, or to individual nodes or edges. And as we'll see in the last few episodes of this sequence, some metadata is critically important to implementing the common graph algorithms. Namely, breadth-first search and depth-first search.

But first! Let's talk about graph structure and metadata.

Variations on the Theme



In the last episode I showed this graph. <next> And in that telling of the story, I showed code that looked like this.

Like most design decisions, this is a tradeoff. On one hand, you decouple nodes from any awareness of edges. This is a good thing for testing the node class, since it doesn't have extra information to complicate things.

On the other hand, keeping the edges and nodes separated means we have to pass around more stuff. Like, whenever you want to get the edges related to a node, you'll need the relevant edge sets on hand.

This is really a topic for a later software engineering course, but I wanted to bring it up now so you at least see that there's usually (maybe always) more than one reasonable way to do something. So let me show you one more way of doing it, and then I'll get into the metadata stuff.

Coupled Data Types

```
struct edge {
  node* end;
  string type;
};

struct node {
  string name;
  set<edge*> edges;
};
```

A different way of coding this up is to have the edges refer to their endpoints, and nodes to refer to the edges that start there, so they're coupled. And since we're talking about metadata, let's use a single set and use metadata to distinguish different edge types.

In this example, the `edge` has a `type` member, which is just a string. We could call it anything we want, and in this example maybe I should have called it `disposition` or `attitude` or some such. Anyway, that `type` field is metadata because it adds additional information to the edge.

Coupled Data Types

```
struct edge {
  node* end;
  string type;
};

struct node {
  string name;
  set<edge*> edges;
};
```

Edge type has references to the destination node.

Node type has references to all outbound edges.

...we store the edges in each node, and edges refer to their ending nodes. In this case, if we have a node, we can use it to add and access our edges.

Whereas in the earlier *decoupled* version, the nodes and edges were totally separate. If you have a node, you'd still need some other data structure to add and access the related edges.

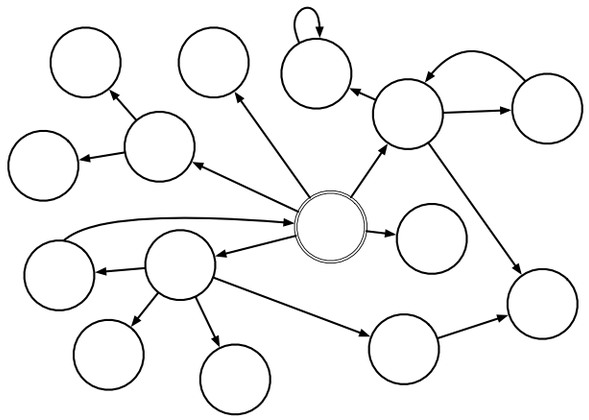
Coupled Data Types

Access edges via their originating node.

```
int main() {
  node* ravi = make_node("ravi");
  node* janis = make_node("janis");
  node* dieter = make_node("dieter");

  ravi->edges.insert(make_edge(janis, "enjoys"));
  ravi->edges.insert(make_edge(dieter, "enjoys"));
  janis->edges.insert(make_edge(ravi, "enjoys"));

  janis->edges.insert(make_edge(dieter, "dislikes"));
  dieter->edges.insert(make_edge(janis, "dislikes"));
}
```

This is how you might use coupled data structures. Use the node, like janis here, to access its edge list, and add new edges, like saying that janis enjoys playing with Ravi but doesn’t like playing with Dieter. I'm only showing adding here, not accessing, though you'd use similar syntax for that.

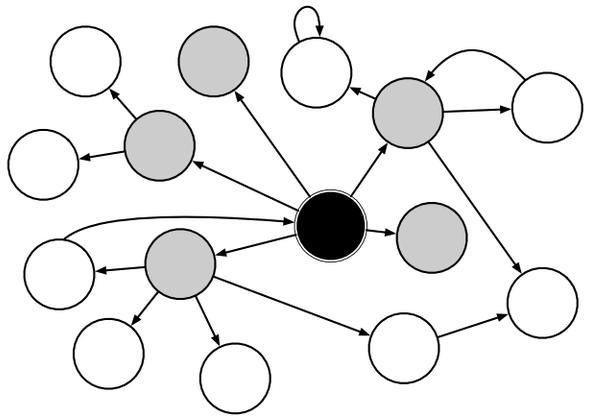
Metadata



OK, now on to metadata.

When you have a graph, one of the main things you'll do with it is traverse it. We've traversed linked lists by scrolling through it, and binary trees by doing some kind of walk: pre-order, in-order, or post-order, remember that? With graphs, we can traverse it in two main ways: breadth first, and depth first.

Node Metadata

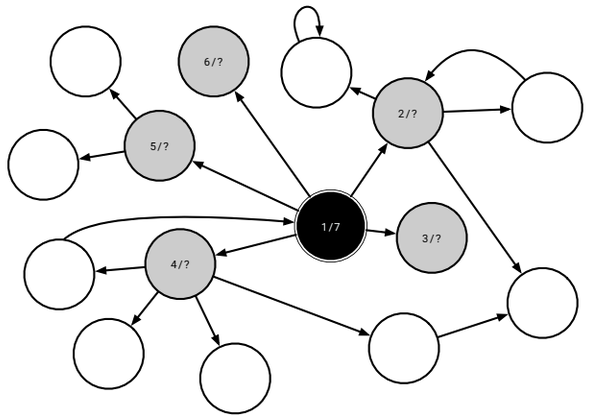


As we traverse the graph it is helpful to maintain a record of where we've been, and one way to do this is by annotating our objects with metadata.

Here's a snapshot of traversing our graph using a breadth-first search, visualized with colors. In this picture we just finished visiting the first node, and the gray ones are up next. And the colors I'm using here are sort of canonical: white nodes are unvisited, gray nodes are currently being visited, and black nodes have been completely visited already.

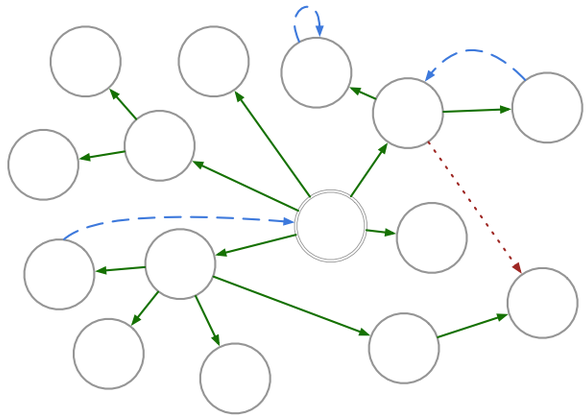
Color is one of the metadata values.

Node Metadata



You can also have discovery and completion times. That is, at which step is a node turned gray, and then black.

Edge Metadata



We can also apply metadata to edges to record the edge's role in a traversal. I’m just briefly showing these now and waving my hands a bit. We'll talk about this more in a later episode.

When you run a depth-first search on a tree, you can classify each edge you traverse based on where you've been already. This is how you can detect if you've got a cycle, for example. That blue dashed edge there is called a 'back edge', which is the metadata we attach to the edge when we follow it and end up at an already discovered node.

Decoupled Metadata

```
struct node {
  string name;
};
```

```
int main() {
  set<node*> black;
  map<node*, int> discovery;

  node* das_node = make_node("howdy");

  // instead of das_node->discovered = clock_time()...
  discovery.put(das_node, clock_time());

  // ... stuff ...

  // instead of das_node->color = black...
  black.insert(das_node); // visit complete
}
```

Earlier we talked about the difference between coupled and decoupled nodes and edges, and that there were tradeoffs involved.

The same thing goes for node and edge metadata. In the homework assignment, we're including the metadata directly in the node and edge classes, so the metadata is coupled. But to show an alternate way of doing it...

<next build>

... you could maintain your metadata separately, like in a set or map. This example shows one possible way for maintaining metadata about nodes without muddying up the node type itself.

Decoupled Metadata

```
struct node {
  string name;
};
```

Node type has no metadata (e.g. color or discovery time)

```
int main() {
  set<node*> black;
  map<node*, int> discovery;

  node* das_node = make_node("howdy");

  // instead of das_node->discovered = clock_time()...
  discovery.put(das_node, clock_time());

  // ... stuff ...

  // instead of das_node->color = black...
  black.insert(das_node); // visit complete
}
```

Here, the node type doesn't have any extra stuff for the metadata that we might or might not use. It only contains the data that directly pertains to it's node-ness.

Decoupled Metadata

```
struct node {
  string name;
};
```

```
int main() {
  set<node*> black;
  map<node*, int> discovery;

  node* das_node = make_node("howdy");

  // instead of das_node->discovered = clock_time()...
  discovery.put(das_node, clock_time());

  // ... stuff ...

  // instead of das_node->color = black...
  black.insert(das_node); // visit complete
}
```

Use decoupled containers to hold data about nodes.

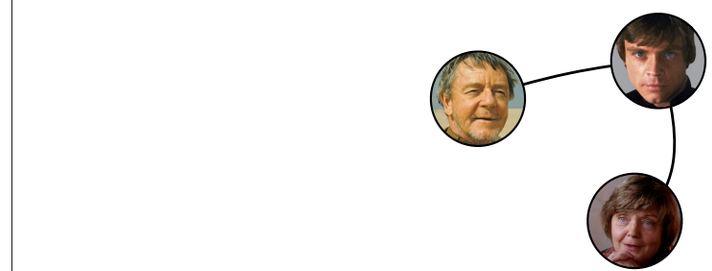
To ascribe metadata to the node, we use `das_node` as a key into other data structures. Like the discovery time map that relates nodes to numbers, and the black set that refers to nodes that are completely visited.

Episode 4

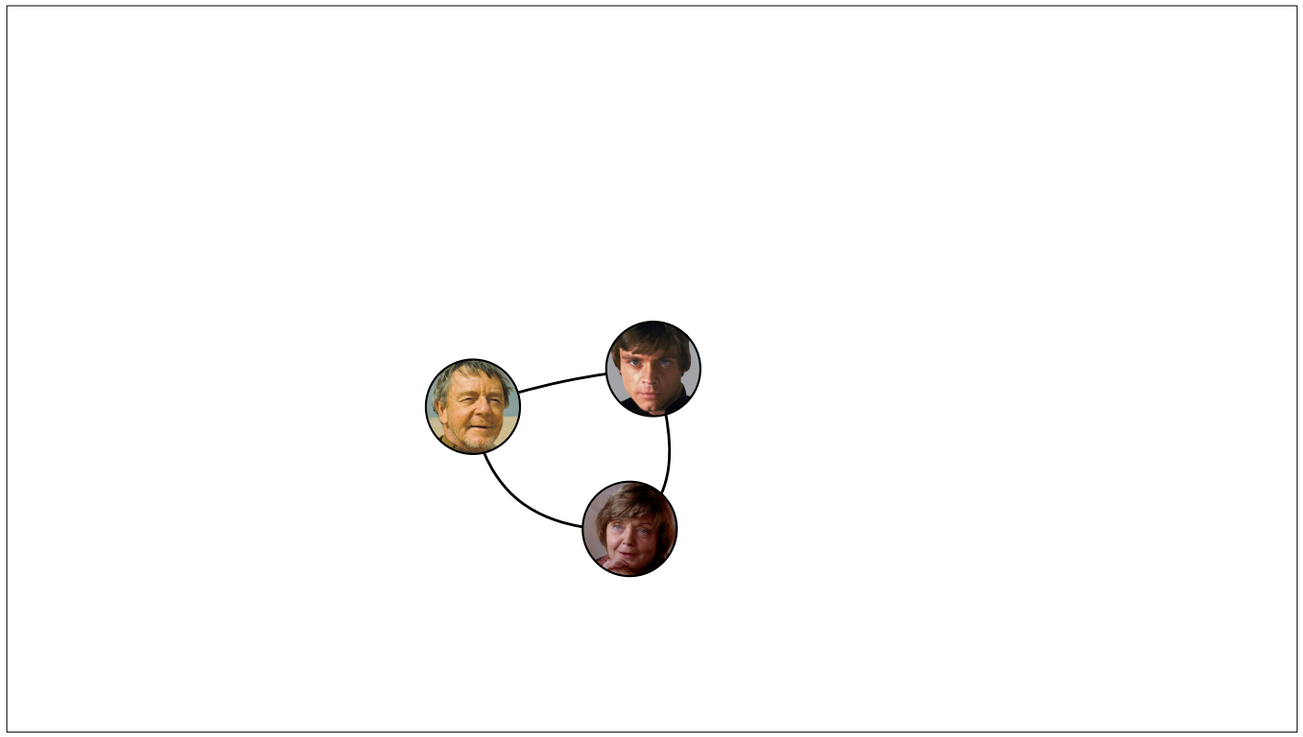
Star Wars: A Social Network

The whole point of assembling graphs is that it lets us do computation. Can I get from A to B? How many steps away are they? What's the shortest path? The longest path? Starting from node A, which nodes can we get to by only following two edges? Three? Starting from any node, what is the node that has the most reachable nodes in two steps? All of this is fine and dandy, but it is very abstract. We should get concrete. And when we get concrete, the only reasonable thing to do is talk about Star Wars.

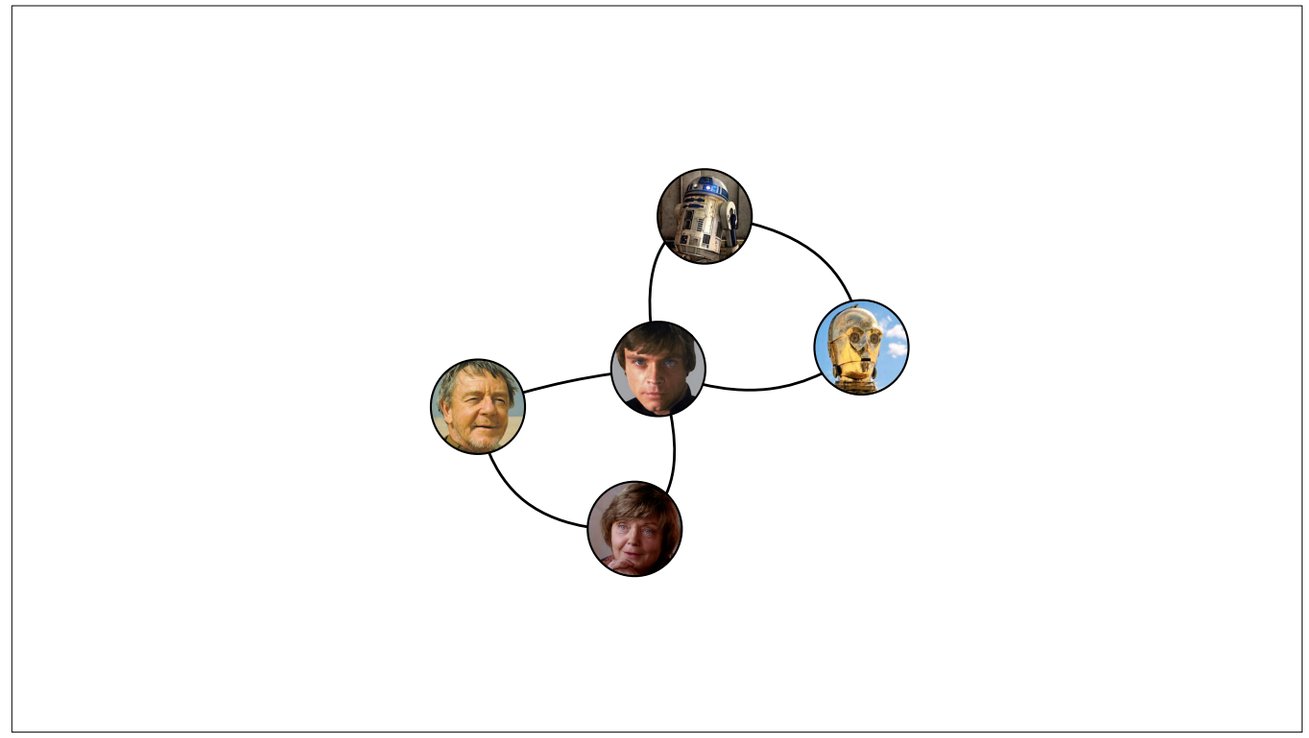
This is Luke. He's in a social network of one. Poor guy.



He lives on Tatooine with his Uncle Owen and Aunt Beru. Notice that the edges are undirected: no arrowheads. In this graph, personal relationships are not directional.

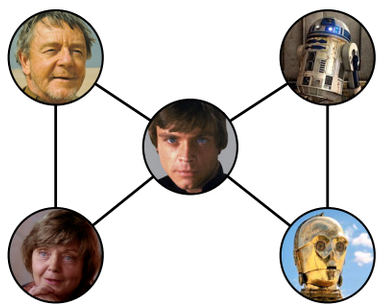


Uncle Owen and Aunt Beru are married, so we should also have an edge that connects them.



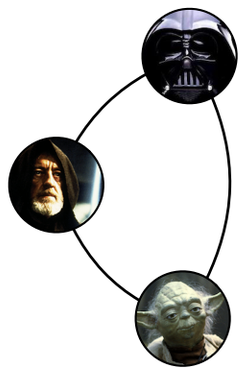
C3PO and R2D2 are buddies, and they are both (sort of) friends with Luke. But not with Owen and Beru, who treat them like toaster ovens.

Using our eyeballs, we can see that Luke is forming the center of this social network. Everyone is connected to Luke, but the path from either droid to Uncle Owen is longer.

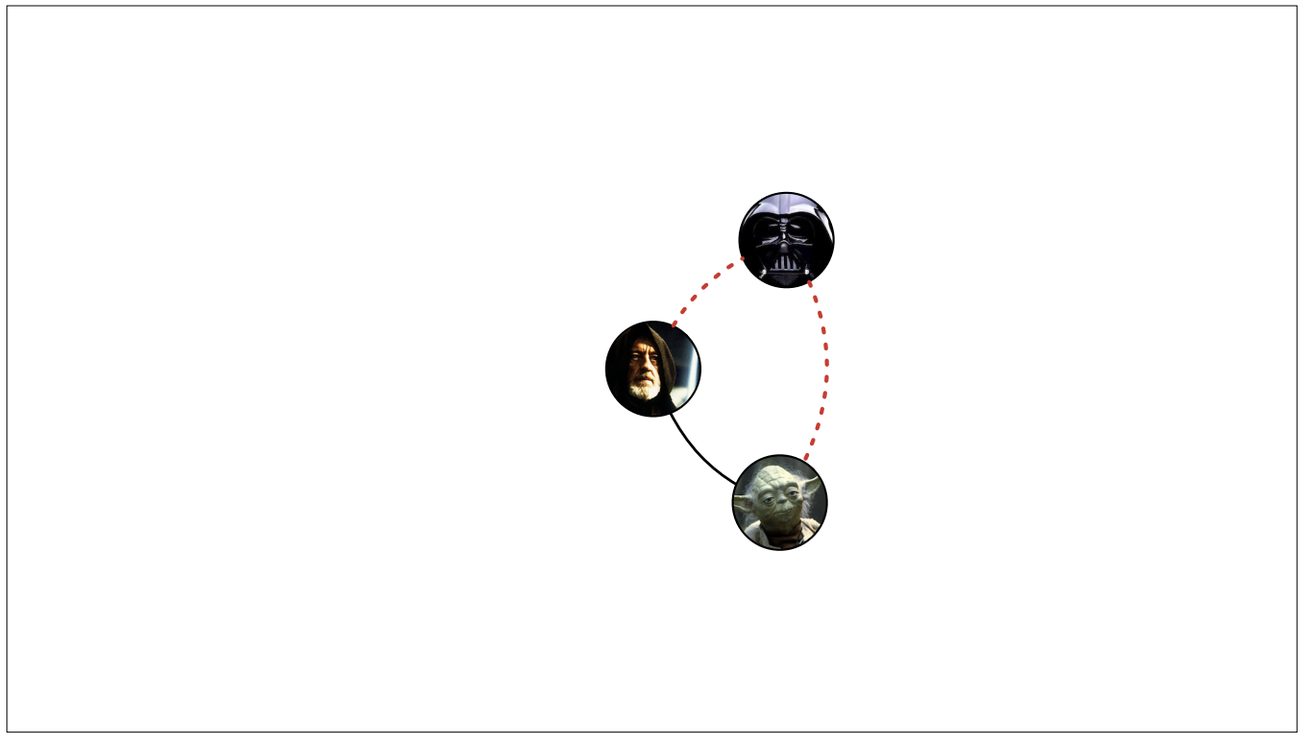


This is the same network. It doesn't matter where the nodes are. The only thing that matters is how they are connected--that's called the graph's _topology_.

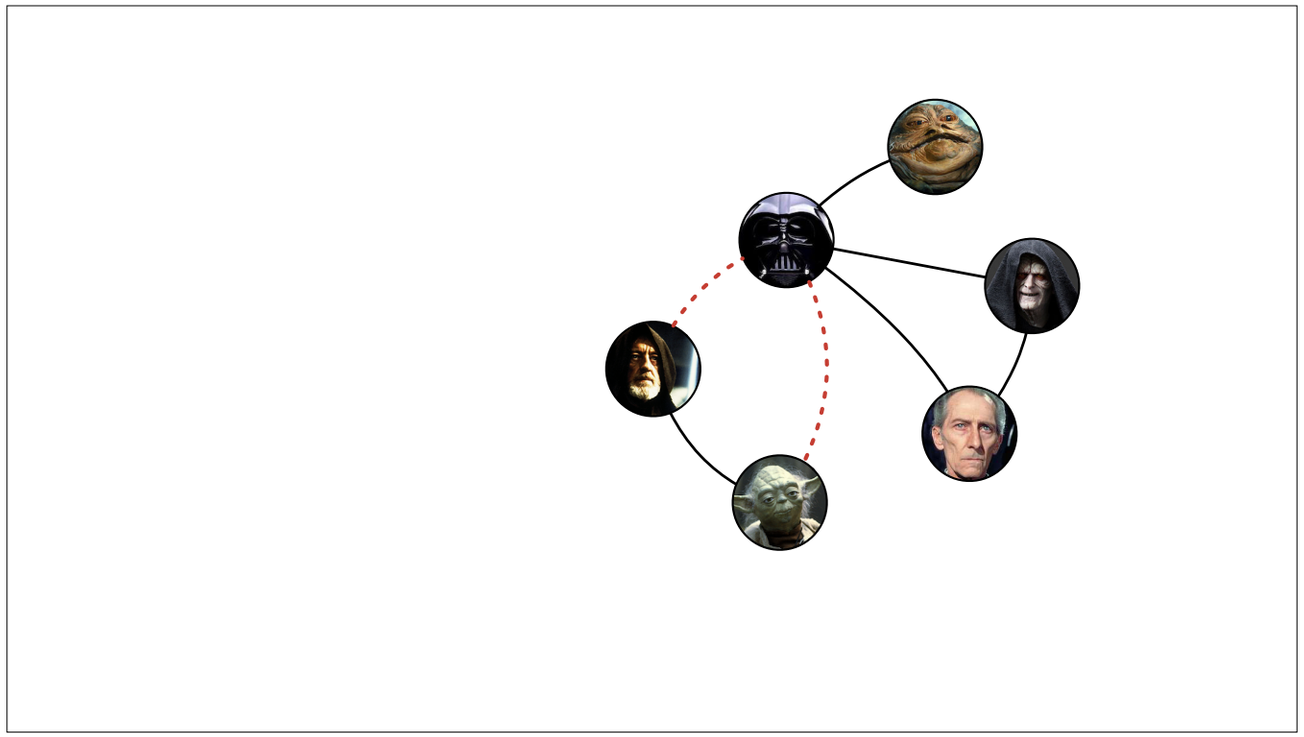
We can build a second graph starting from this guy, Obi-Wan Kenobi.



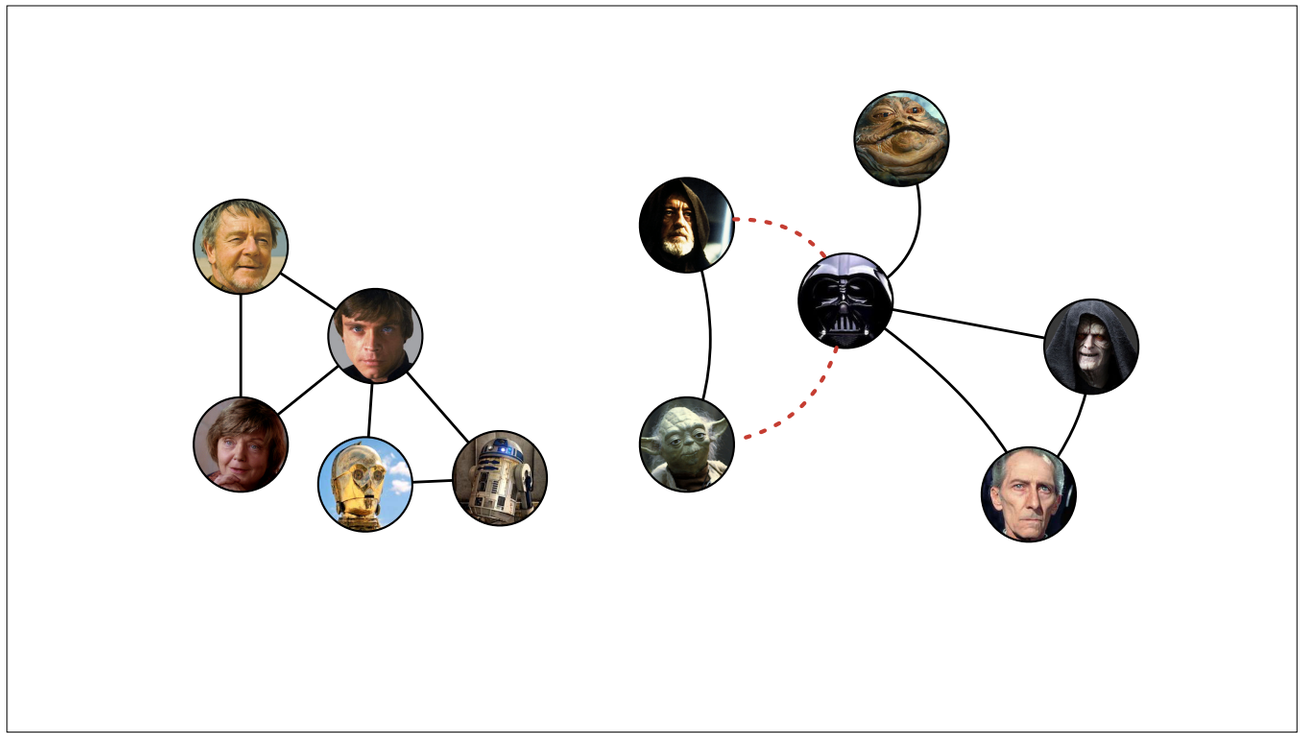
Obi-Wan has a long history with Yoda, and with Darth Vader. They're connected too. Now we are starting to see connections that mean different things. It is complicated, but the connections to Darth Vader indicate opposition.



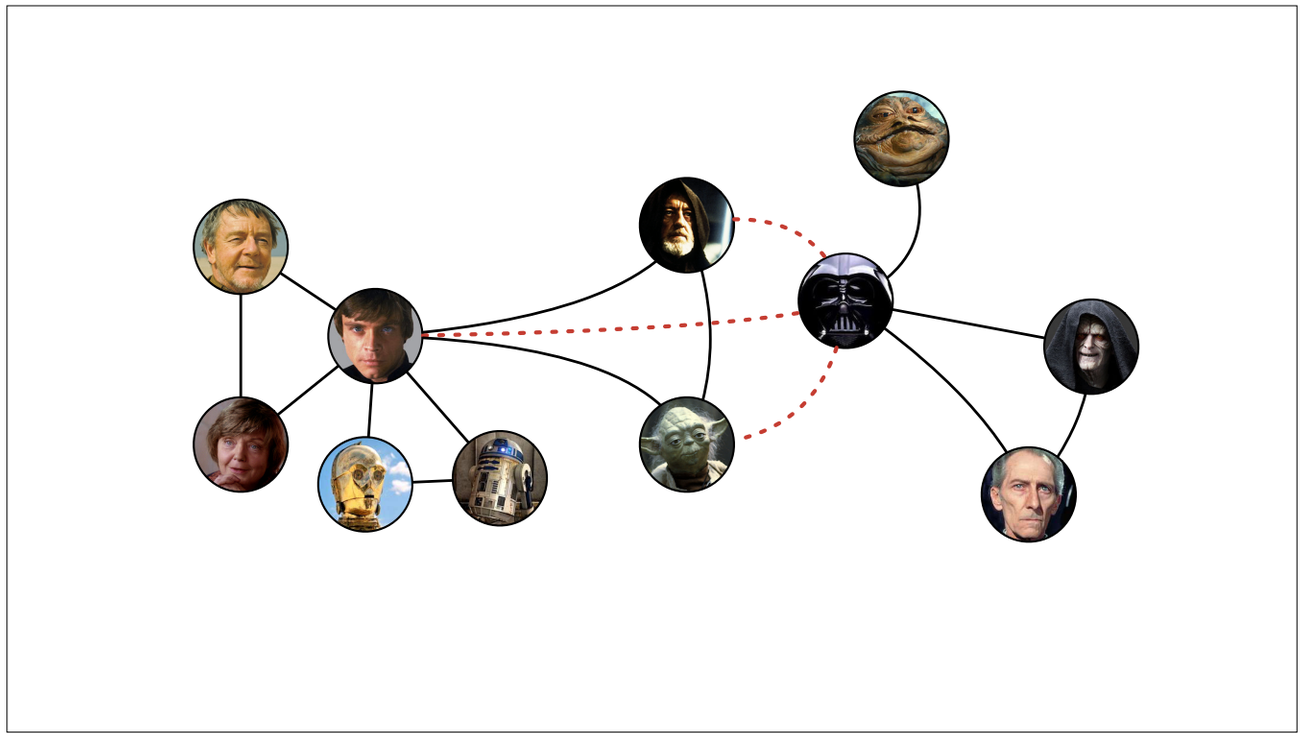
So we have edges indicate different kinds of relationships. It isn't directional (no arrows), but we will need to record the nature of the relationship as hostile. I'll do this with colors. Red dashed for hostility, solid black for friendly.



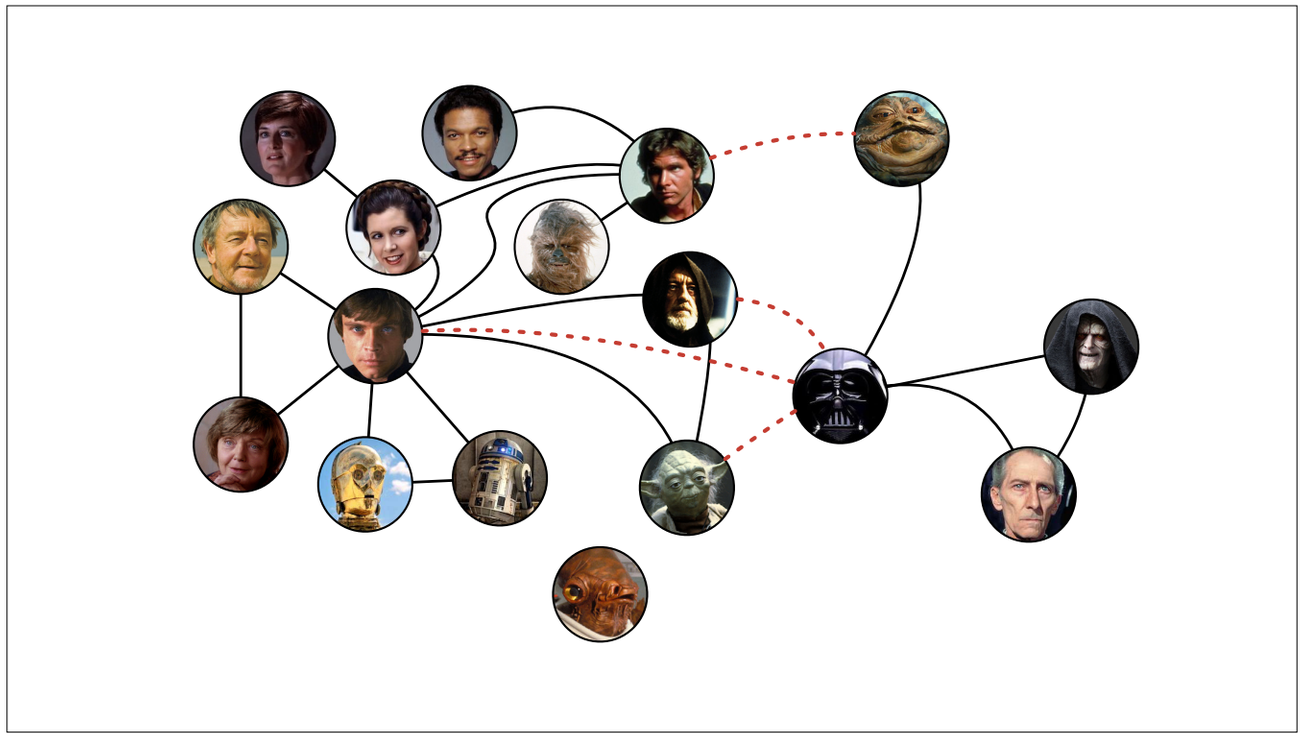
Darth Vader does have some buds, though. The evil Emperor is basically his BFF. Vader struck an odious deal with Jabba the Hutt, and relies on Admiral Tarkin to blow up planets for him.



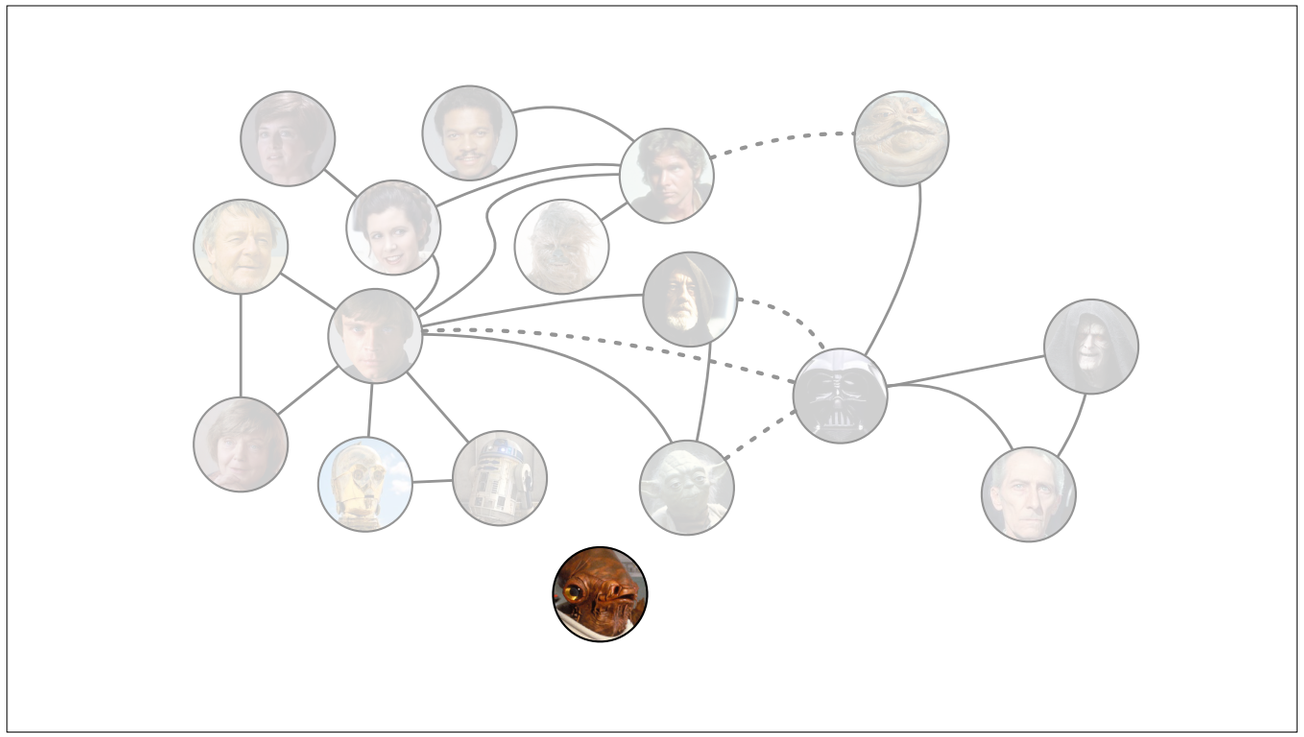
So now we have some of the Luke Skywalker network, and some of the Bad Guy network.



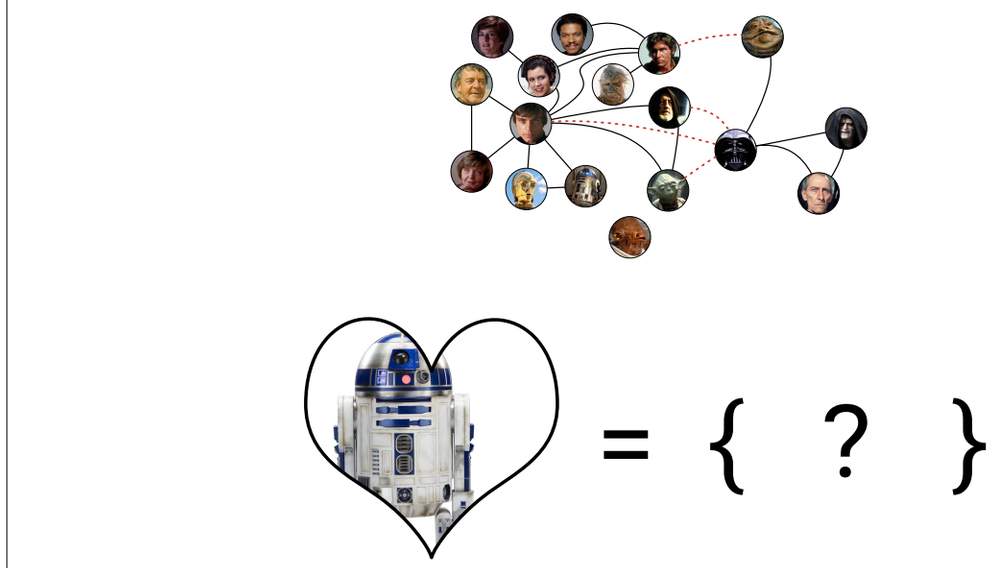
Luke meets up with Obi-Wan and Yoda, and joins the battle vs. Darth Vader.



For grins, we can put the rest of the cast in here too.



Notice how poor Admiral Ackbar is in the graph, but he's not connected to anyone. Aside from being kind of sad, this is actually a legitimate thing. Graphs nodes don't have to be connected to other nodes.



I thought this was about graph algorithms!

It is! I promise!

Lets say we are given the graph, and would like to find all of the people who are connected to R2D2 in a friendly way. This means following friendly edges (no hostile edges) and forming a set of nodes.

<next build>

How do we do this?

There are a few ways. But they are all going to use one of two primary strategies: a depth-first search or a breadth- first search. We'll do the depth-first search first.

Episode 5

Depth-First Search (DFS)

This episode is 100% concentrated depth-first search, using the social network from Star Wars.

```
DFS(node):
  mark node gray
  visit node
  for each edge e related to node:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark node black
```

The pseudocode for a basic depth first search that uses recursion is like this. You could do this in a non-recursive way as well, but we're doing it recursively.

You start the process with some node---we're going to explore starting from R2D2 because we're interested in finding all of his friends.

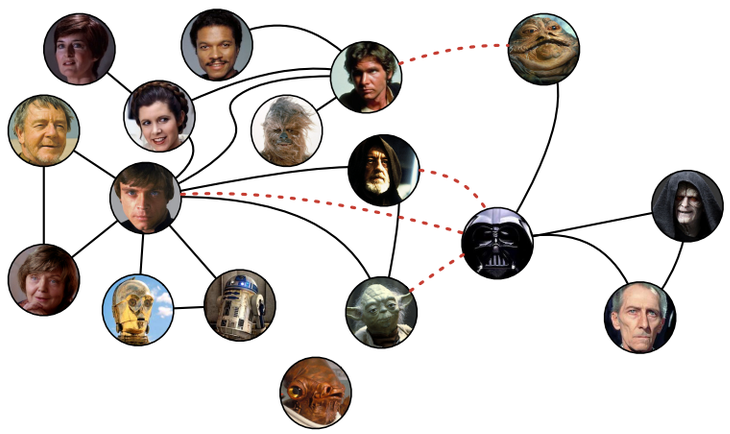
First thing to do is mark the node Gray. That's metadata that you'll read to determine if the algorithm is working its way through that node or not.

Then, make a list of adjacent nodes by looking at our node's edges. Since we're only interested in his friends, we will filter out the edges that represent hostility.

For each edge, get the node on the other end of the edge. If it is currently undiscovered, and by that I mean it is colored White, recursively call the DFS algorithm on it.

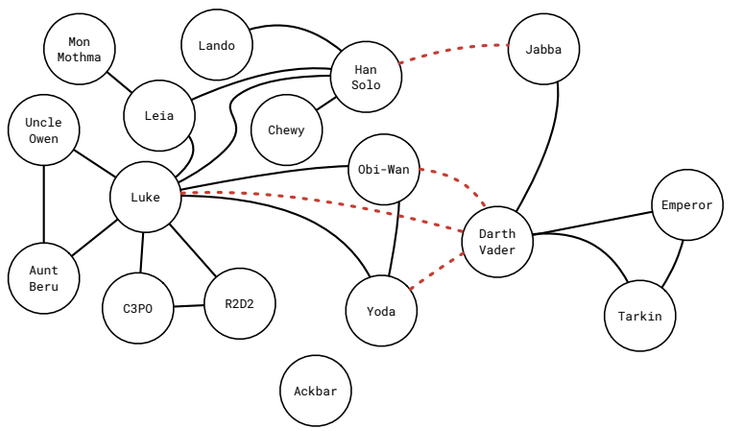
Since you've done so much recursion already in this class you probably already know this, but I'll say it anyway. When you issue that recursive call to DFS on another node, the current incarnation of DFS just waits for it to complete. So when the recursive call comes back, we might have done a whole lot of work, which will all be written down in the edge and node metadata.

Find R2D2's Friends

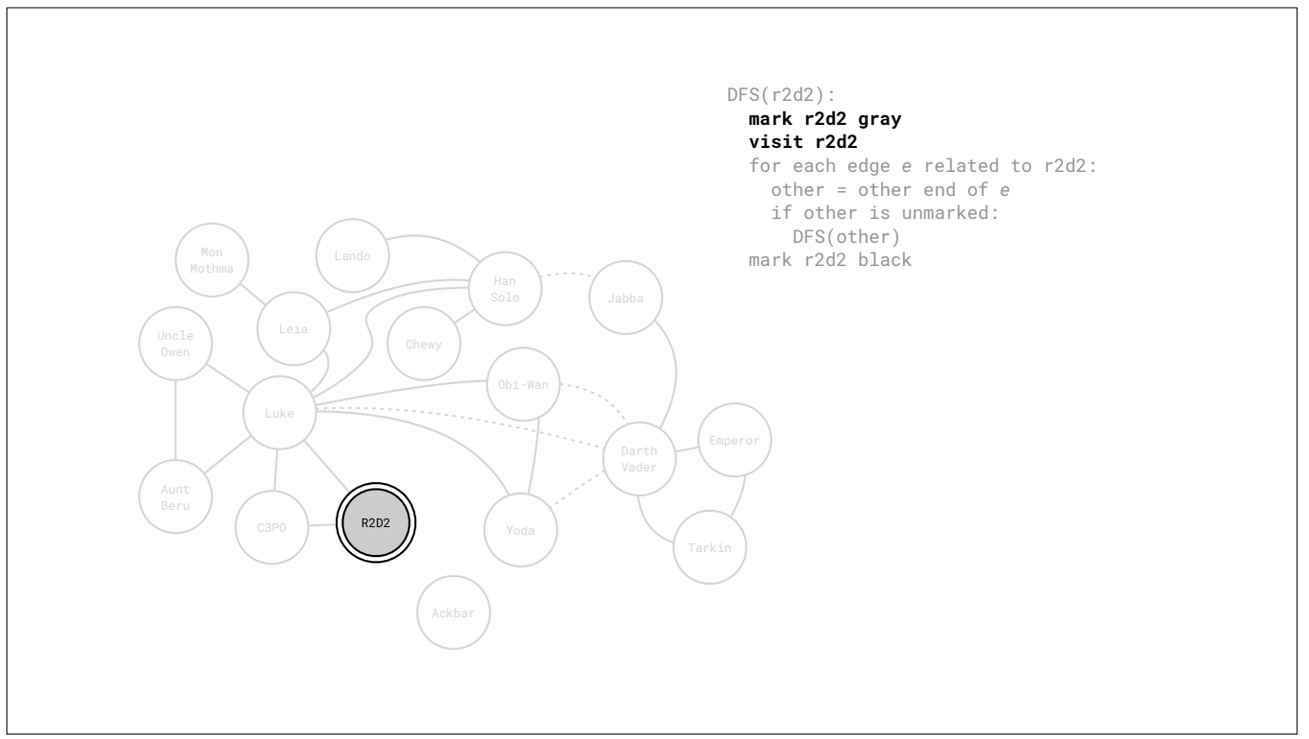


This is the Star Wars social network that we built up in the previous episode. To make it easier to follow I'll take out their mugshots and <next>

Find R2D2's Friends

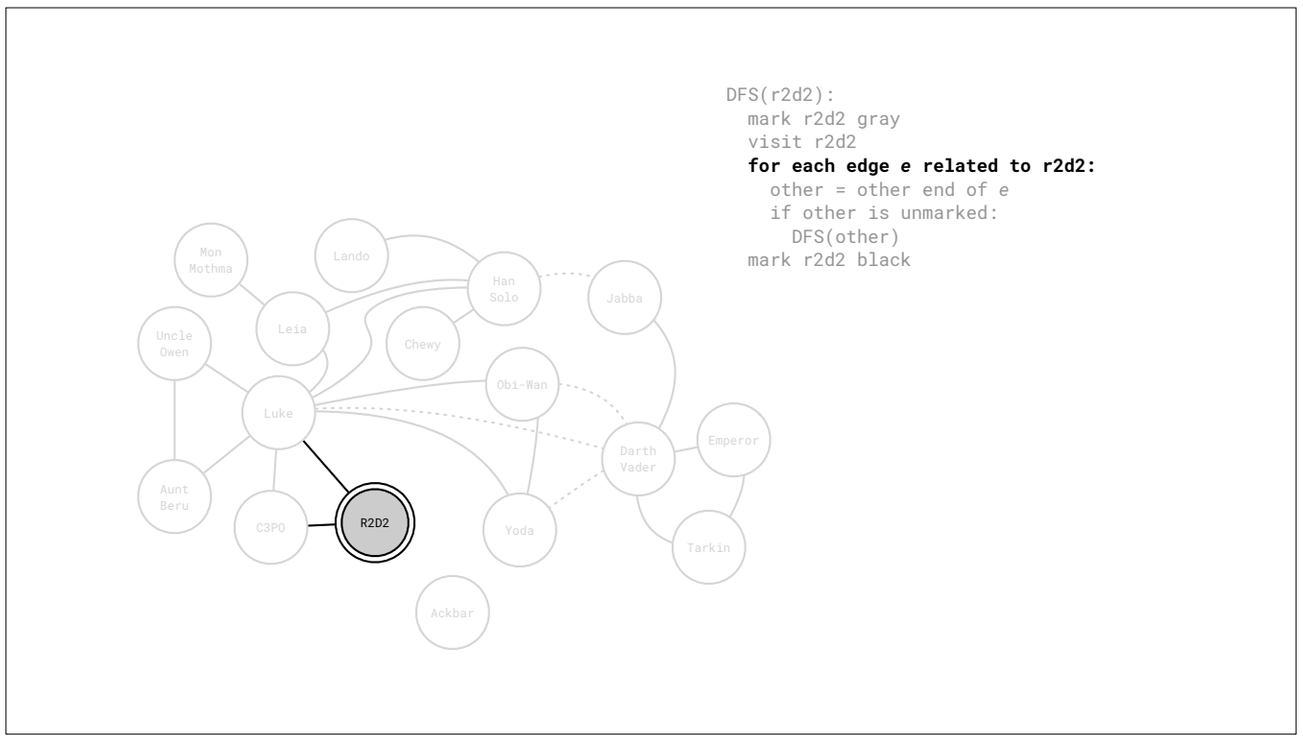


replace them with names instead.

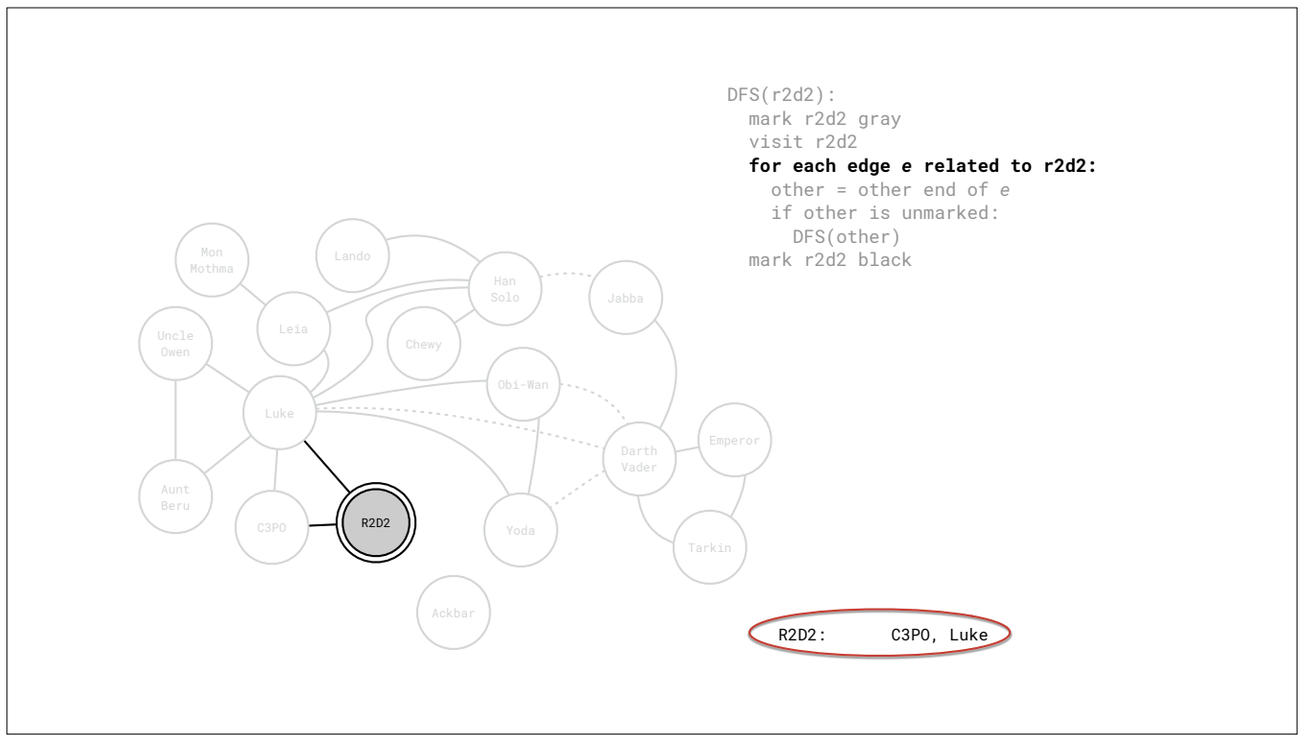


Let's start with R2D2, and we'll draw his node with a double circle so we remember where we started. This is just for our viewing pleasure, you don't actually need to record that as metadata.

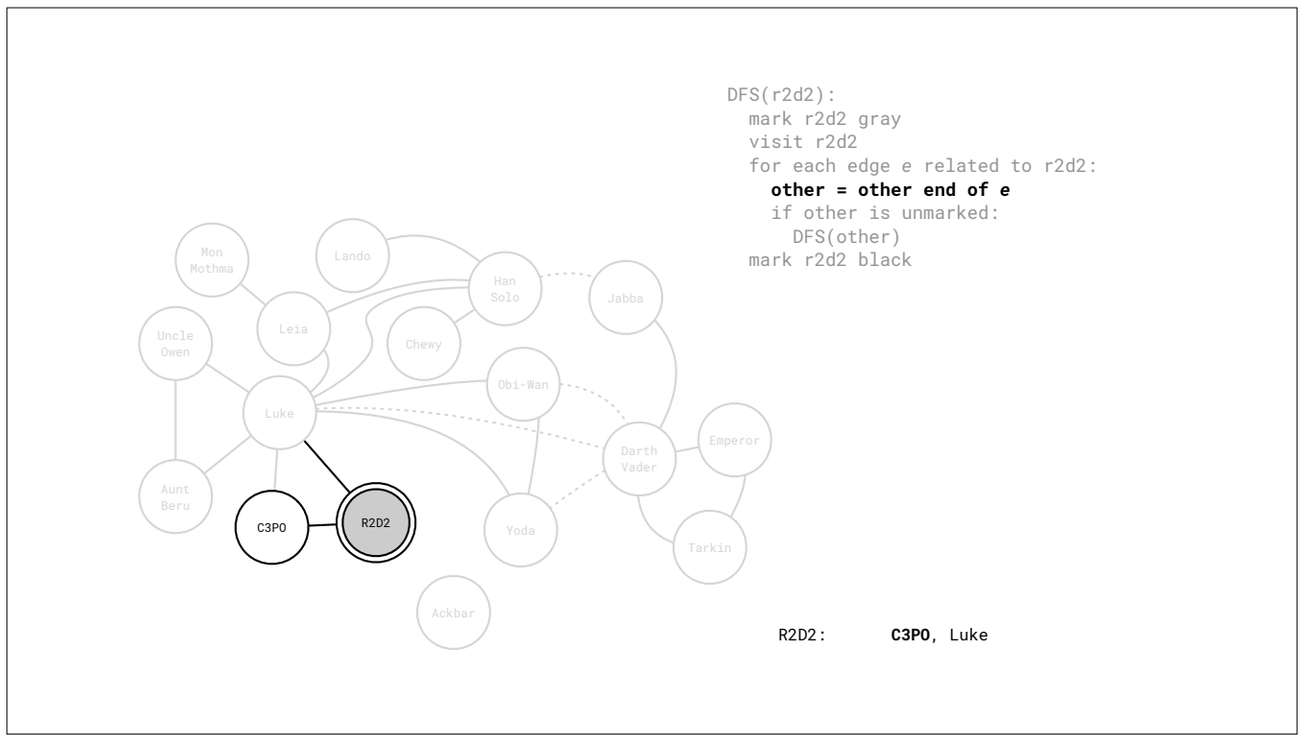
So the first thing we do is color R2 gray, meaning he's been discovered, and we're currently visiting him. If we had any work to do with this node, like if we wanted to print to console, or any other computation, this is where we'd do it.



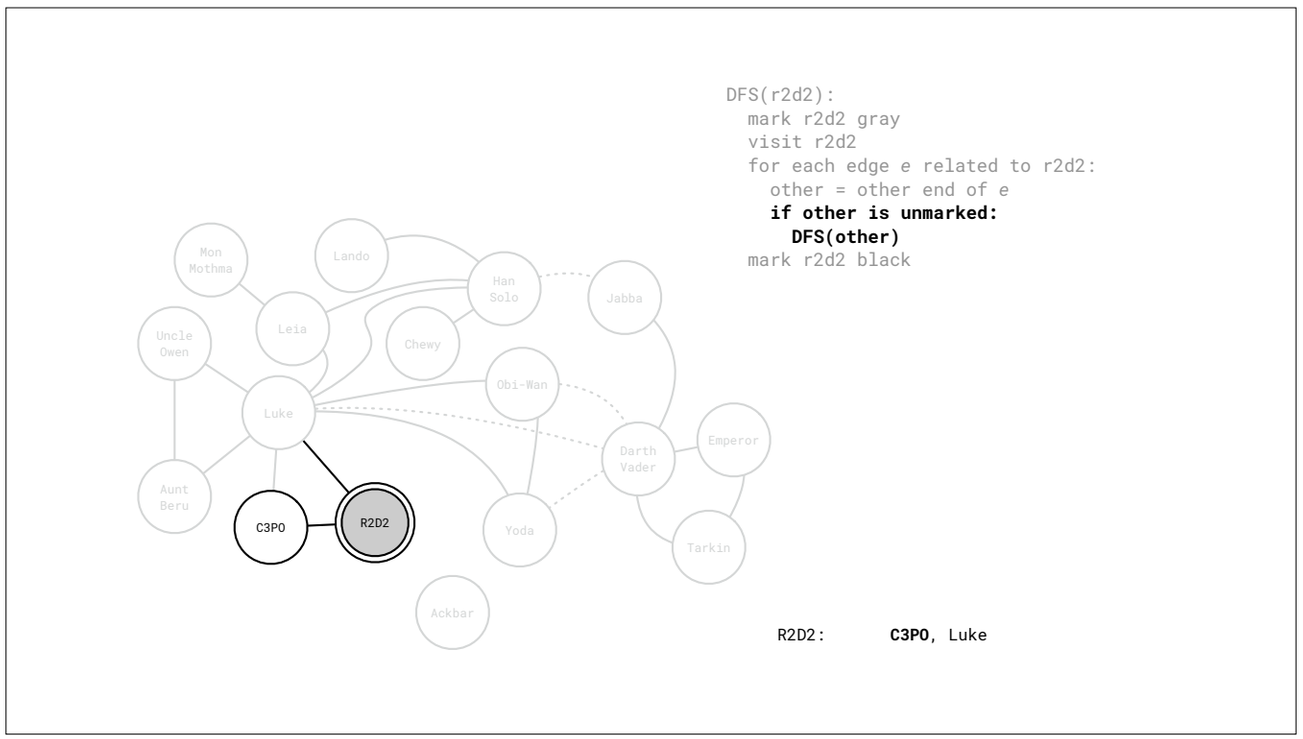
Now that we've marked R2 as discovered, we start iterating through the edges adjacent to his node. We'll just make a list, and there's only two. One leads to C3PO, the other to Luke.



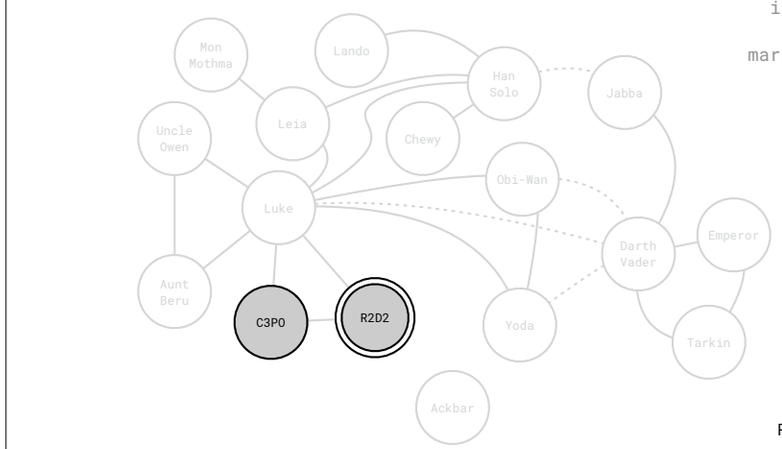
Since we're going to do those recursive DFS calls, I want to make it easy to see on the slide where we are with each incarnation of the DFS call. I'm making a list of edges to process, not nodes, but for clarity's sake I'm going to write them down with the name of the node on the other end. So we've got R2D2 who is related to C3PO and Luke on the bottom right.



OK, let's enter that loop now that we have our list of edges to traverse. The first one we get has C3PO at the other end, so we get a reference to him. I'll boldface the edge that a DFS frame is currently on.



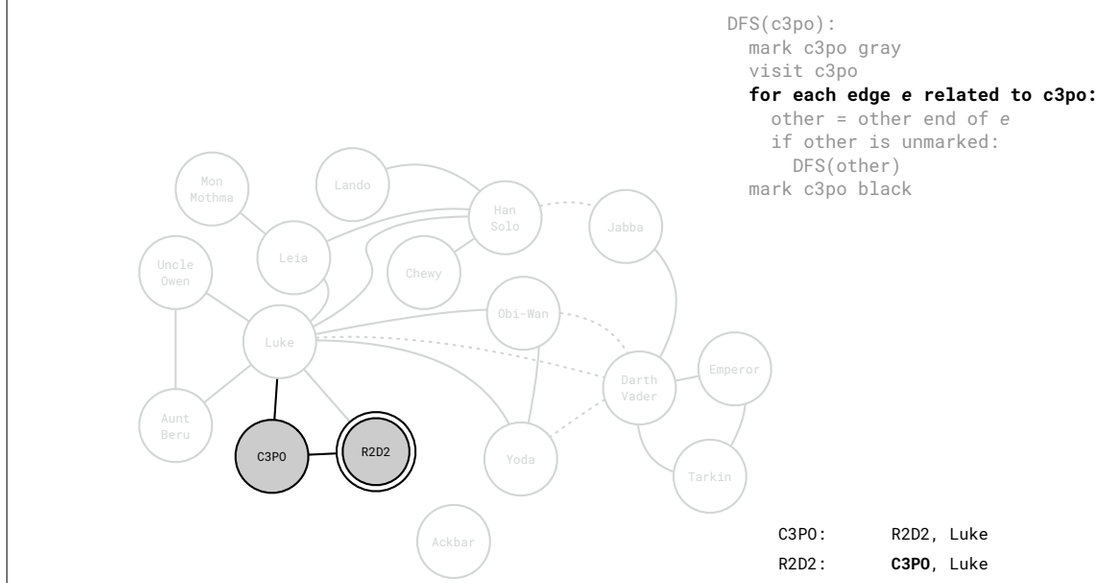
The next thing we do is look at the other node's color. C3PO is still white, so it is unmarked. Because of that, we're going to recurse, calling DFS with C3PO.




```
DFS(c3po):
  mark c3po gray
  visit c3po
  for each edge e related to c3po:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark c3po black
```

R2D2: **C3PO**, Luke

So we recurse using C3PO, leaving the R2D2 frame. First thing we do is mark 3po gray, and visit it.

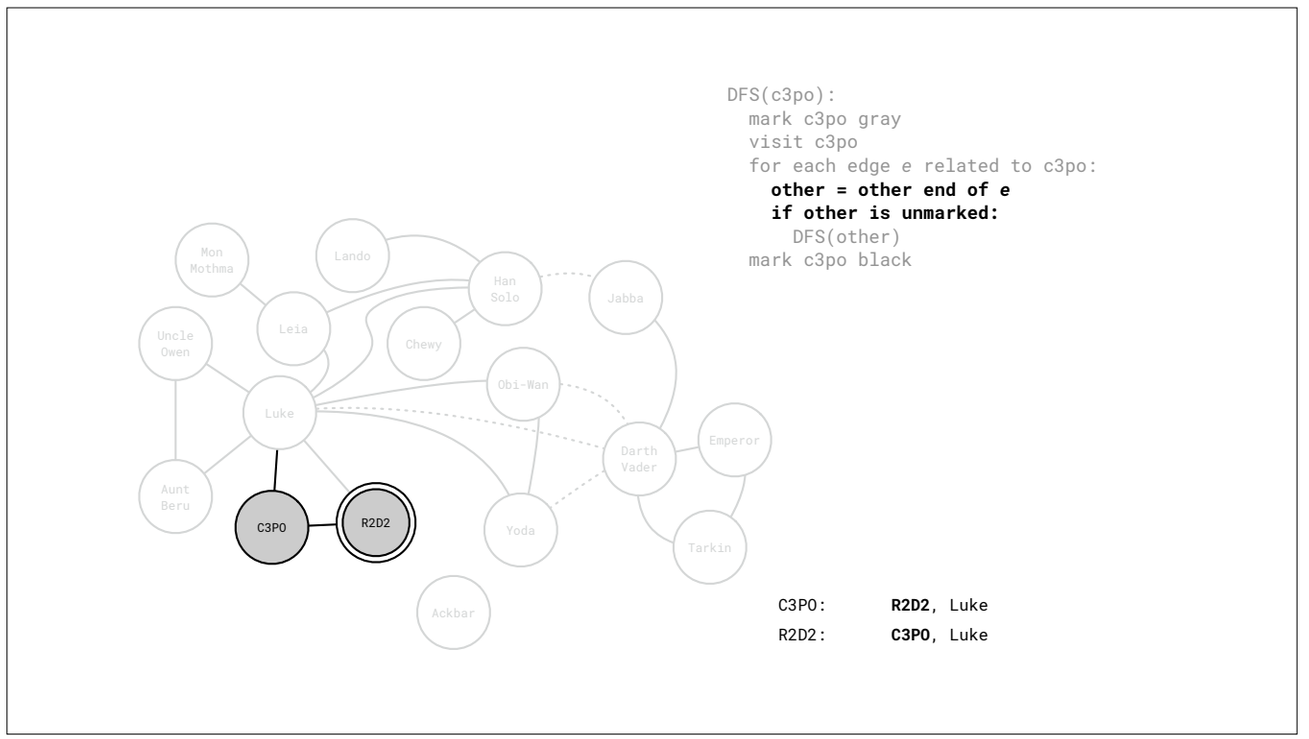



```
DFS(c3po):
  mark c3po gray
  visit c3po
  for each edge e related to c3po:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark c3po black
```

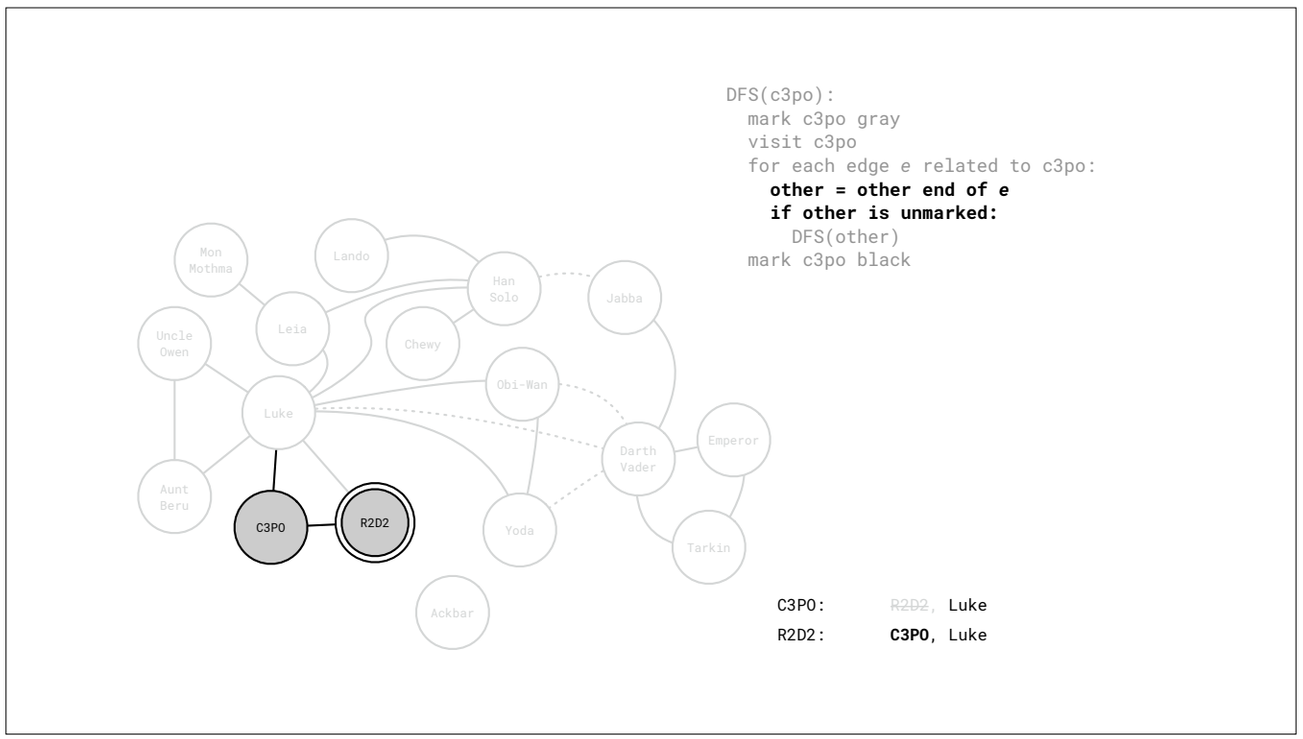
C3PO: R2D2, Luke

R2D2: **C3PO**, Luke

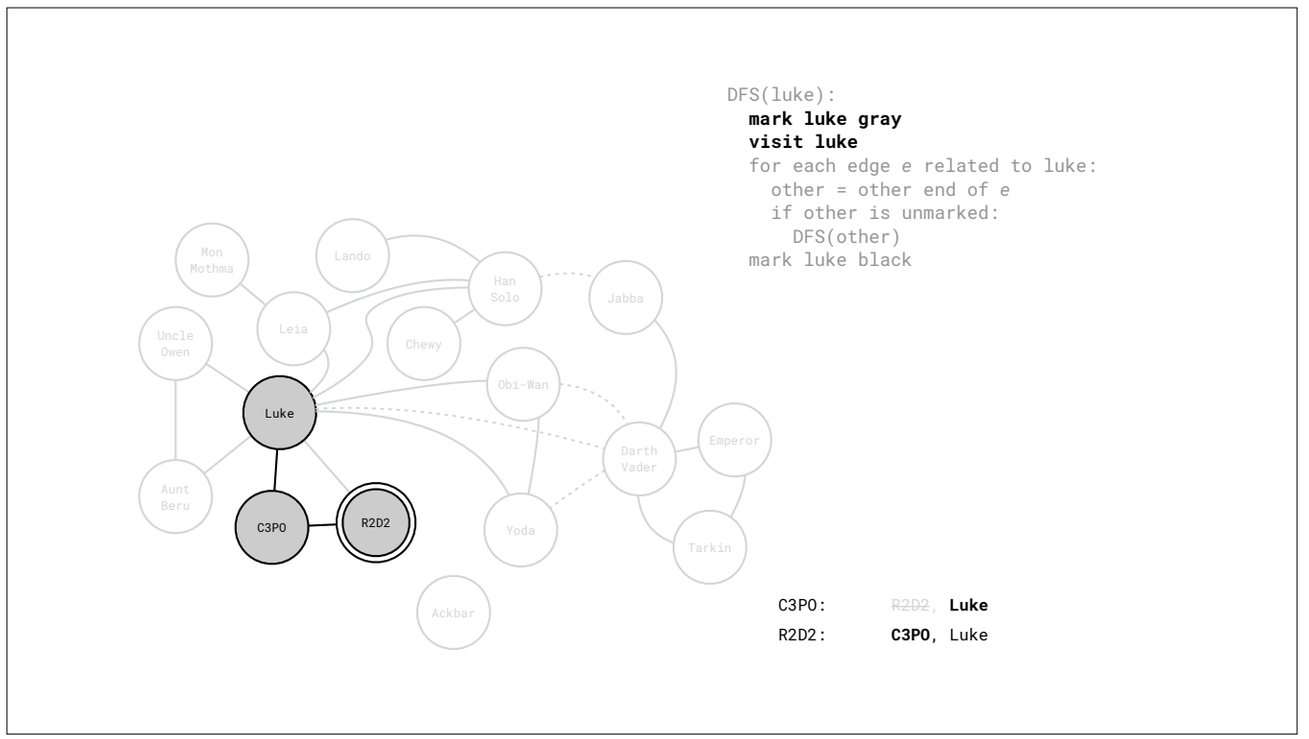
Then, just like earlier, we make a list of edges that are adjacent to our input node. C3PO is related to R2D2 and Luke, so we form another edge list, which I'm putting in the bottom right corner.



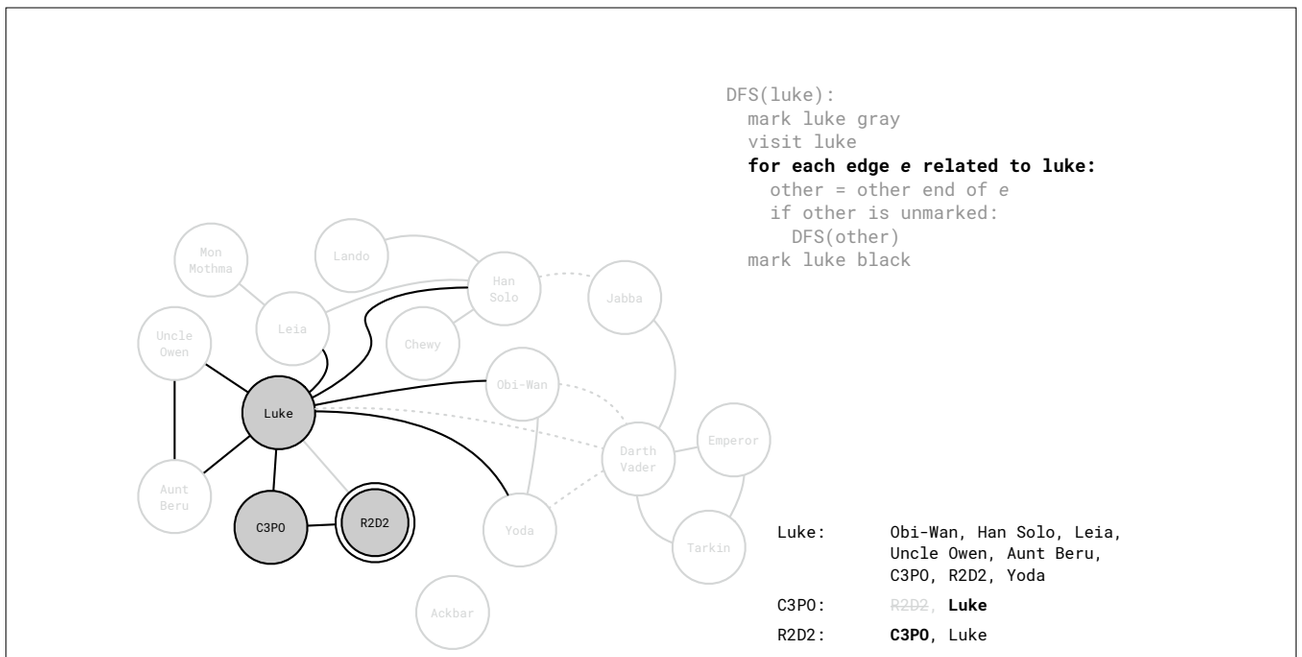
Now we've discovered threepeo, so we start iterating through the edges adjacent to him. The first item sends us to R2D2. This is interesting because once we look at R2D2's metadata, we see that he's gray, which means he's marked. So we then _don't_ recursively do a DFS on R2D2.



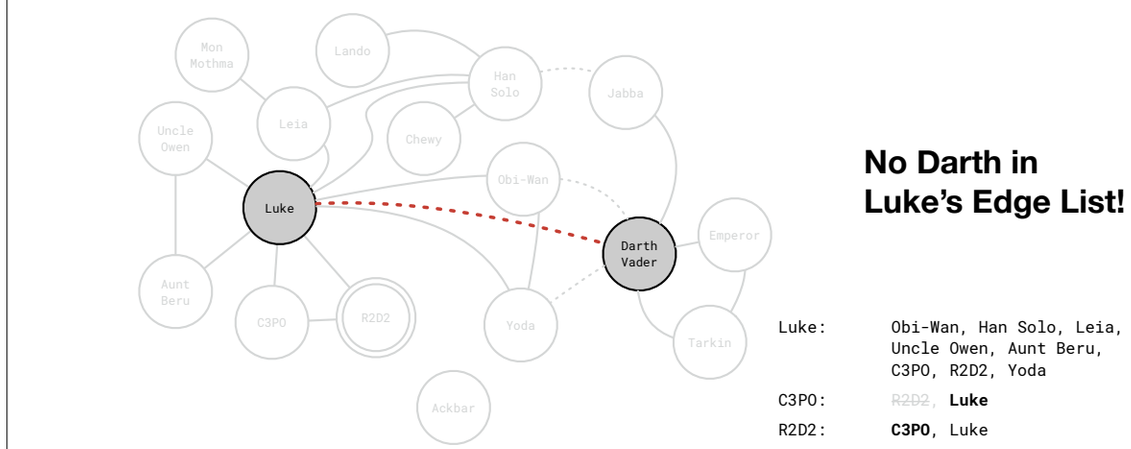
So C3PO is done looking at R2D2, so I'll cross him off in the list, and we can proceed to the next edge.



... which leads us to Luke! We mark Luke gray, visit his node in whatever makes sense, and move on to his edge list.



Which is extensive! He's like that person on LinkedIn with 500+ connections and you're like, how can you possibly know all these people?



No Darth in Luke's Edge List!

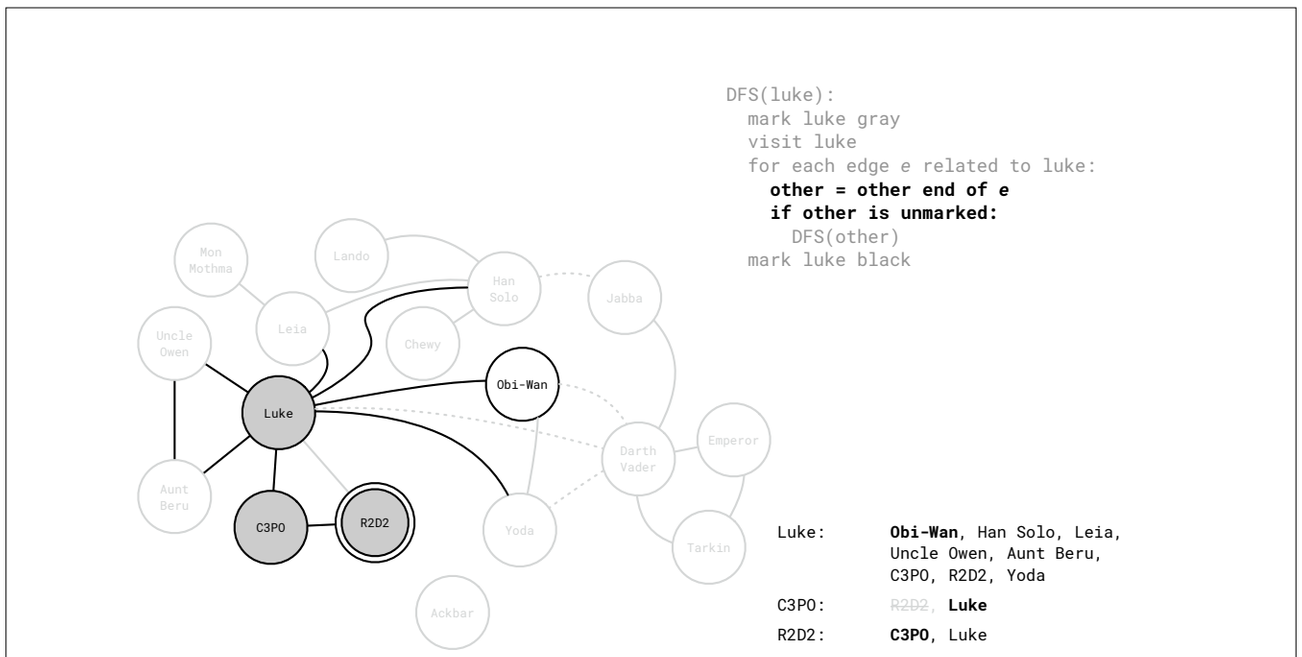
Luke: Obi-Wan, Han Solo, Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda

C3PO: ~~R2D2,~~ **Luke**

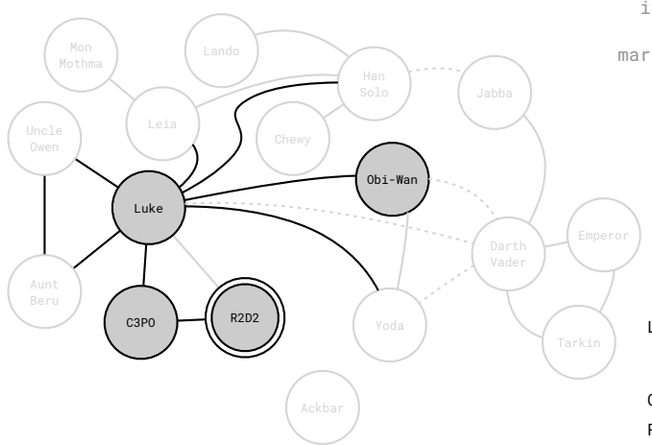
R2D2: **C3PO**, Luke

Now, you know that Luke has an edge to Darth Vader, but why didn't we include that in his edge list? Well, we're only considering _friendly_ edges. You can do this in several ways, like having different edge sets, or have a single edge set but have each edge marked up with metadata that describes its nature. At any rate, the pseudocode is the general form of a recursive DFS, and like I've said a hundred times already, graphs are usually one-offs.

So from now on, consider all those dashed edges to be filtered out. They still exist, we just won't treat them as edges that we'll traverse.



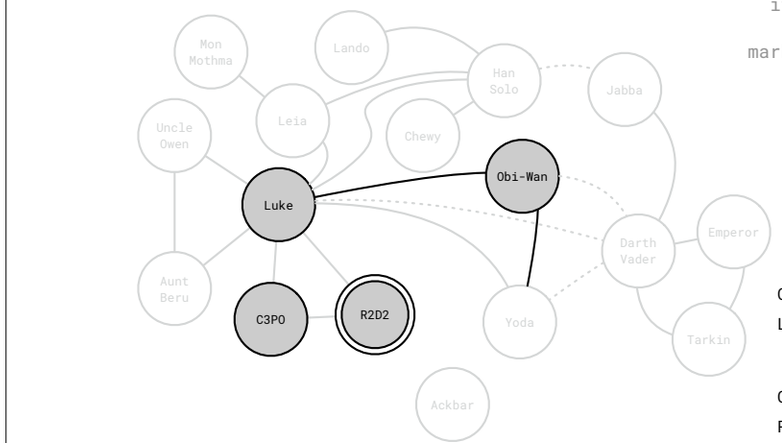
Back to our DFS. The first edge in Luke's list points us to Obi-Wan Kenobi. He's unmarked, so we recurse yet again.




```
DFS(obi_wan):
  mark obi_wan gray
  visit obi_wan
  for each edge e related to obi_wan:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark obi_wan black
```

Luke:	**Obi-Wan**, Han Solo, Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke

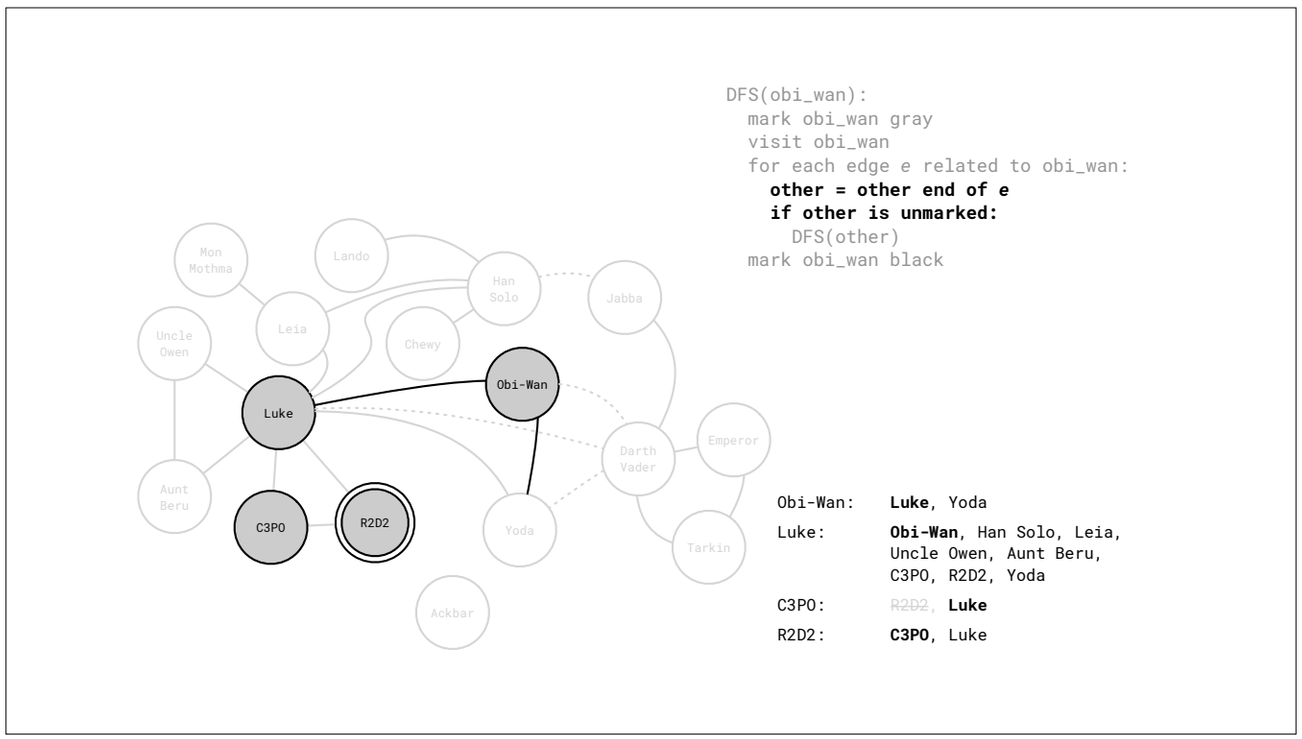
Discover and visit Obi-Wan.



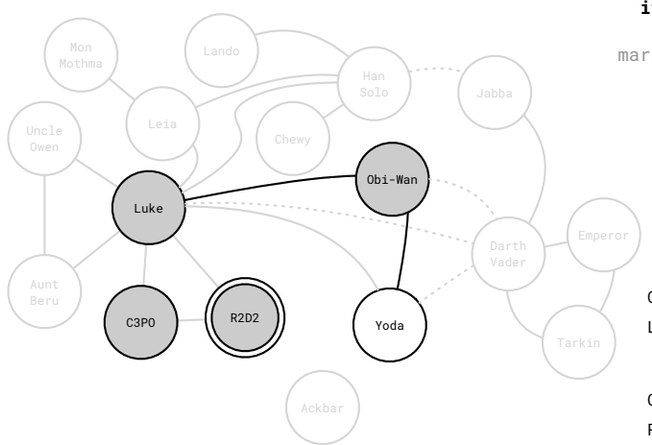

```
DFS(obi_wan):
  mark obi_wan gray
  visit obi_wan
  for each edge e related to obi_wan:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark obi_wan black
```

Obi-Wan:	Luke, Yoda
Luke:	**Obi-Wan**, Han Solo, Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke

Get a list of friendly edges - so don't follow that edge to Darth. It's a trap! We just get Luke and Yoda.



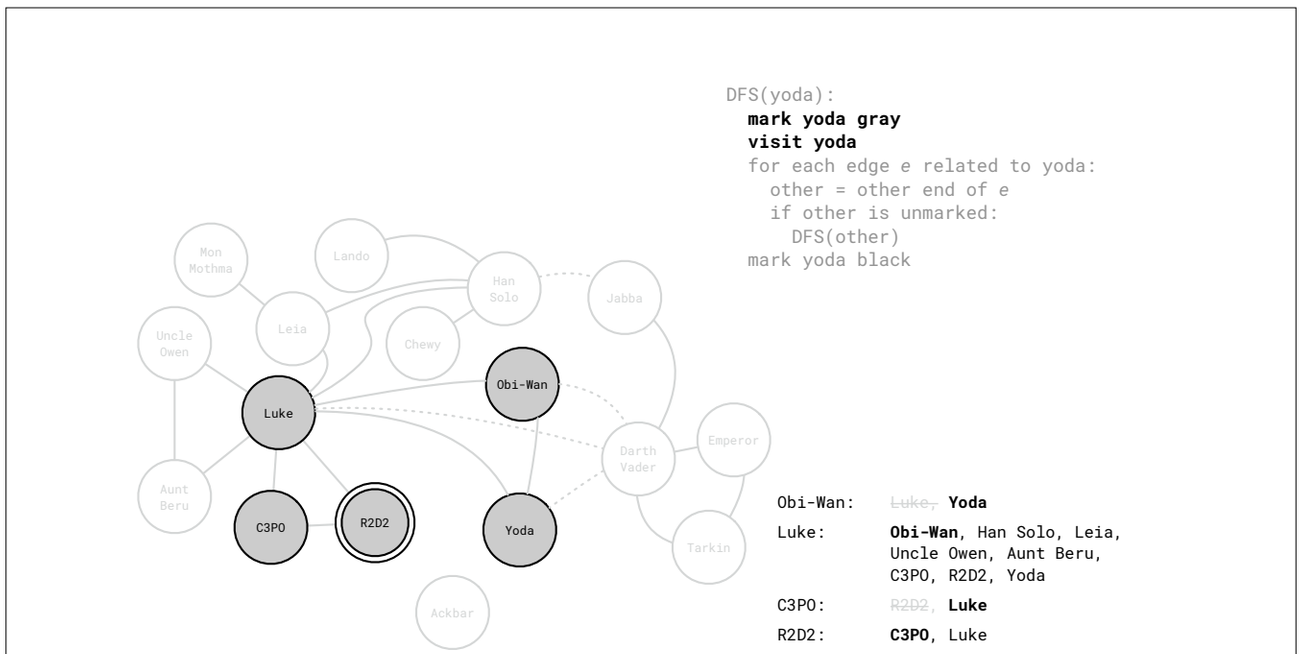
When we look at the first edge's other end we see Luke, and Luke has already been marked gray, so we don't recurse there.



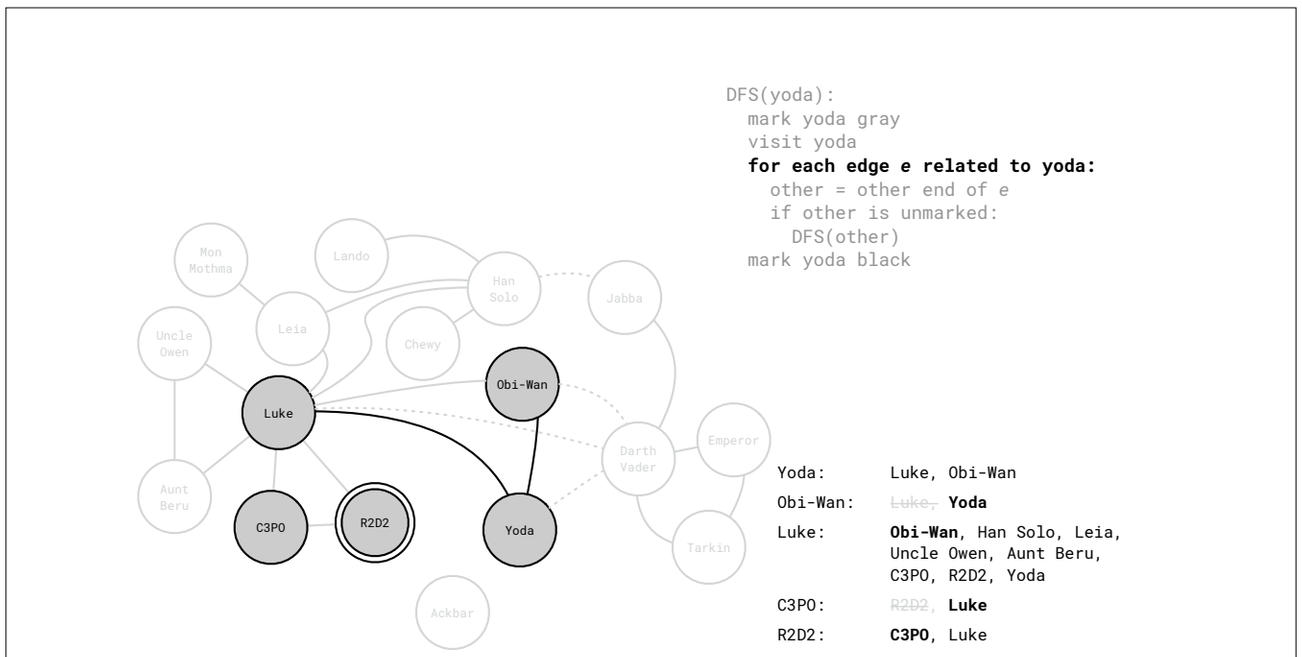

```
DFS(obi_wan):
  mark obi_wan gray
  visit obi_wan
  for each edge e related to obi_wan:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark obi_wan black
```

Obi-Wan:	~~Luke,~~ **Yoda**
Luke:	**Obi-Wan**, Han Solo, Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke

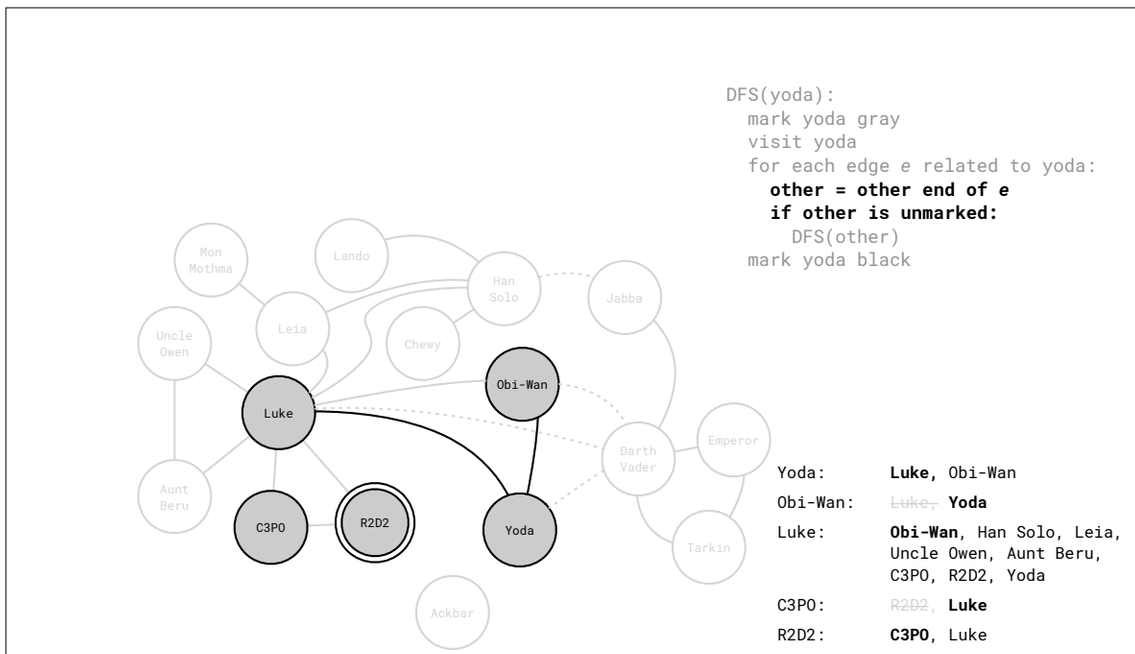
We take the next edge in Obi-Wan's list and find that it points to Yoda, which is unmarked. So we recurse.



Discover and visit Yoda.

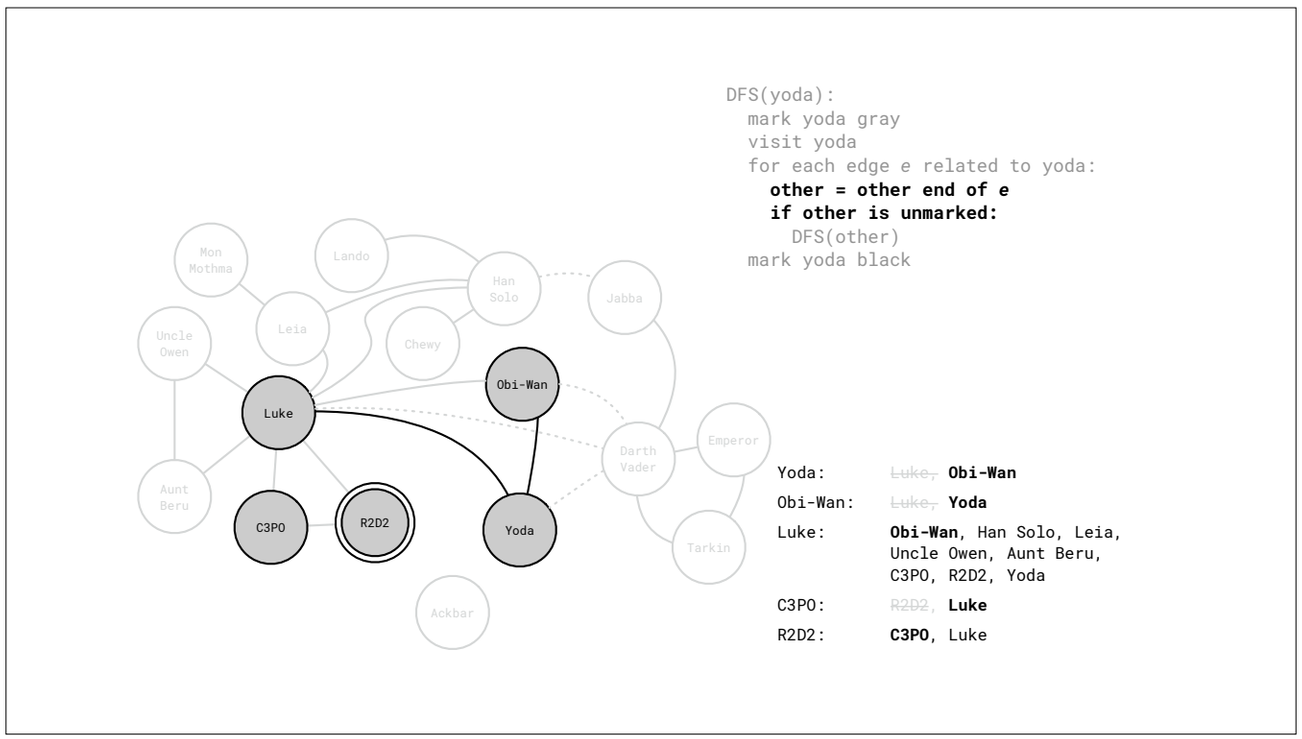


Make a list of all the friendly edges adjacent to Yoda, which is just Luke and Obi-Wan.

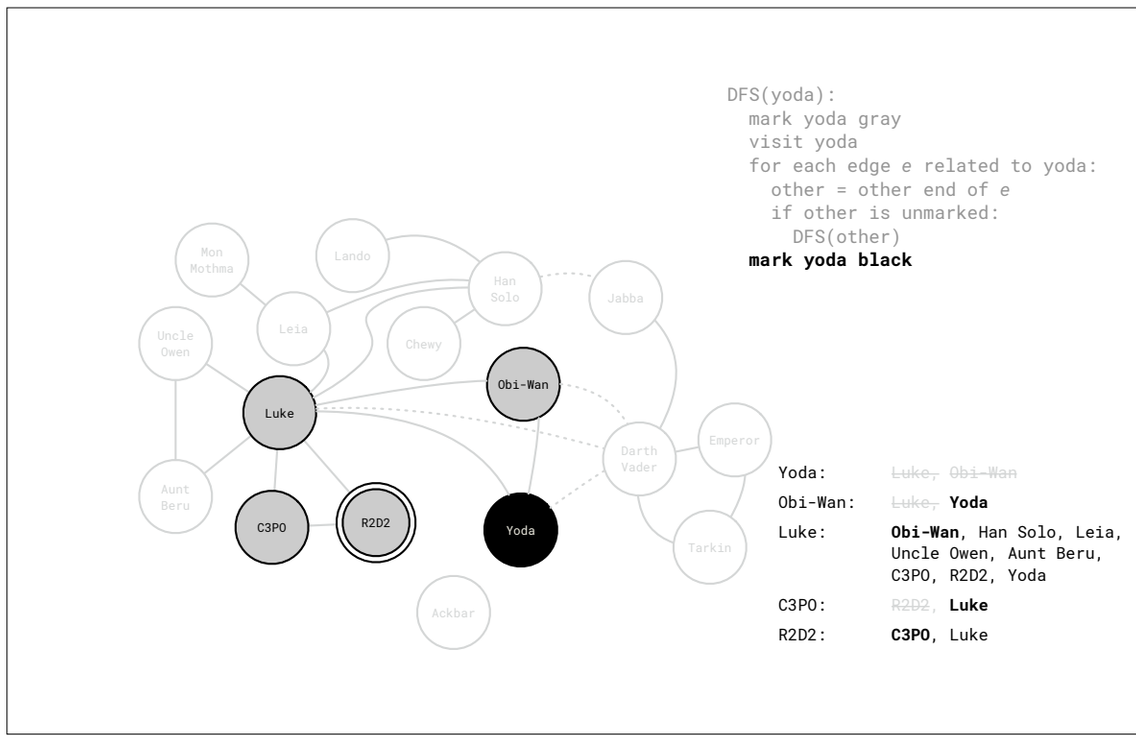


I'm being pretty OCD about this, I know. But I'm about to get to something interesting.

Looking from Yoda to Luke, we see that Luke has been marked already, so skip to the next edge.

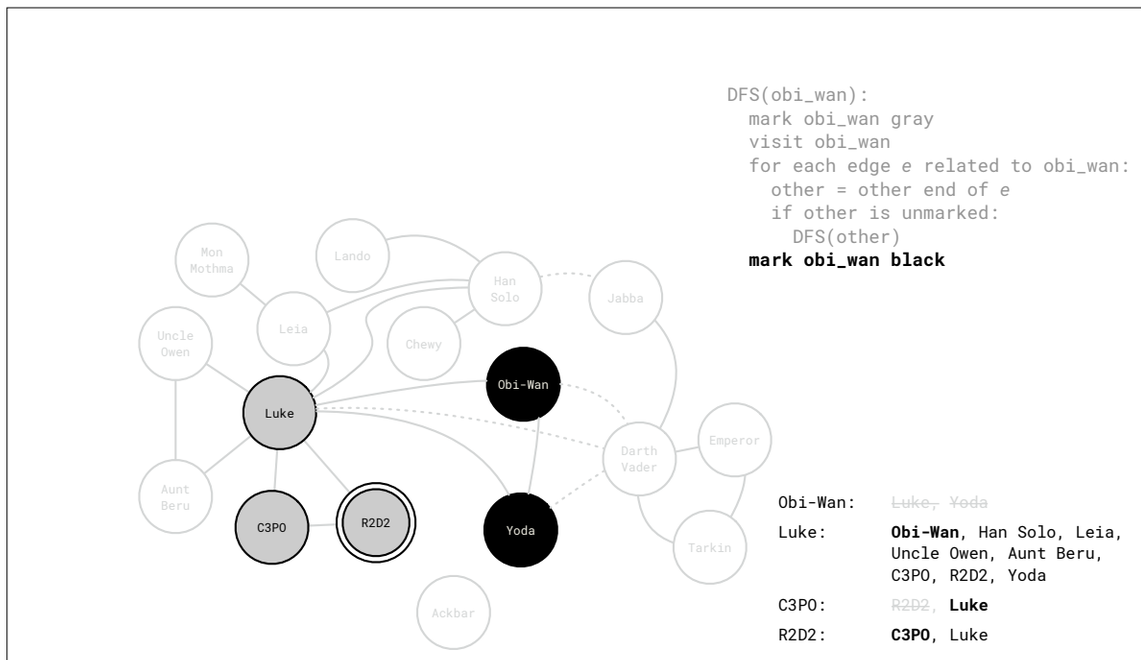


Same thing for Yoda to Obi-Wan.



OK, something interesting has happened! We've exhausted Yoda's list of edges, so we fall out of that loop. And the next instruction is to mark Yoda black. This means we've completely explored Yoda and his outgoing edges.

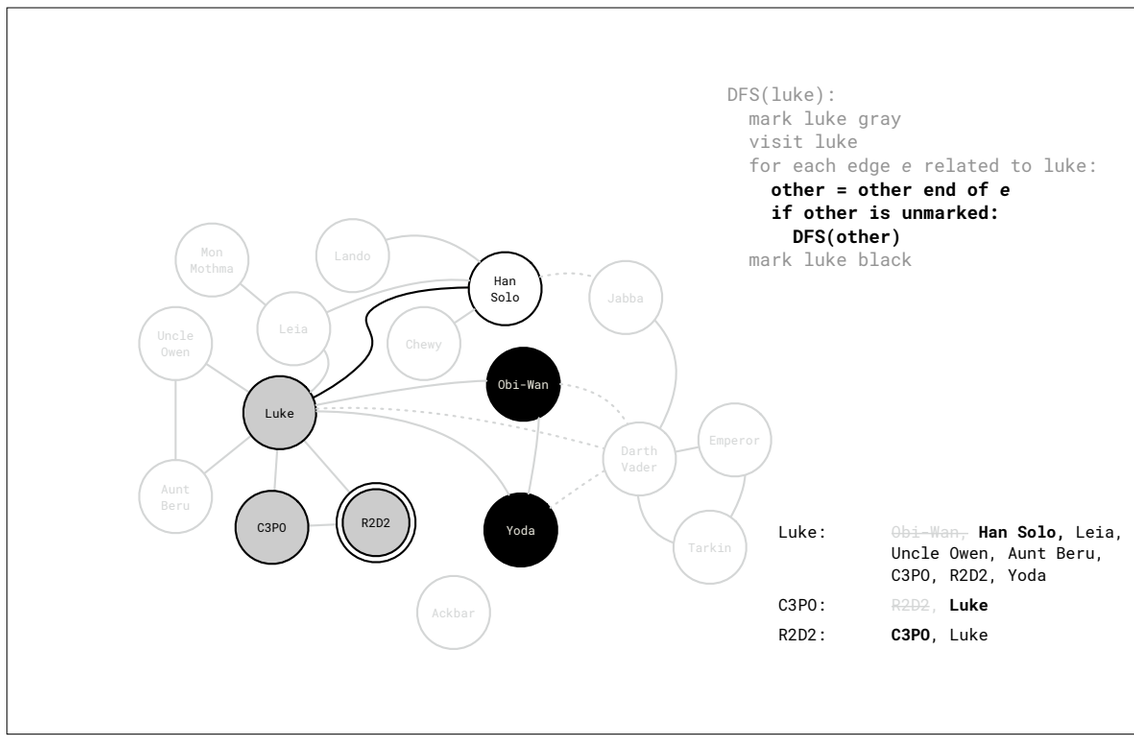
What happens now?



We complete the DFS call to Yoda, which was initiated from the Obi-Wan frame. So control returns to the Obi-Wan DFS. Yoda was the last node adjacent to Obi-Wan, so we fall out of _that_ loop, and mark Obi-Wan black.

I also want to direct your attention to that stack of DFS calls on the bottom right. The Yoda list has already gone away, and you see the Obi-Wan list has been spent. As soon as we return from Obi-Wan, we'll no longer need that either, so we'll pop that from our little visual stack as well.

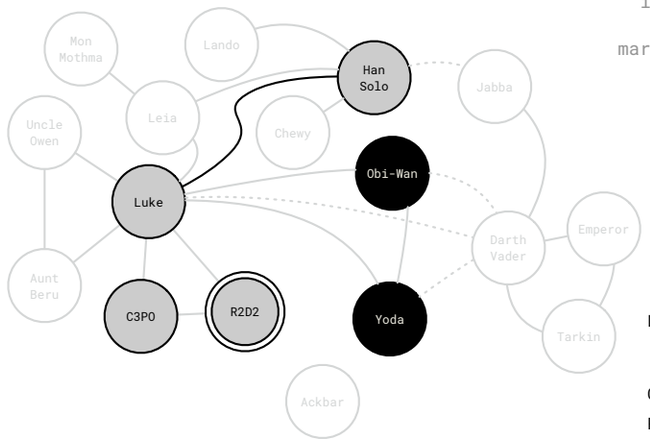
This is intentional. We've been working through the recursive version of DFS, but we could have done this in a single function call, using a stack data structure to maintain the nodes that we'll visit.



Now that we're done with Obi-Wan, we return from whence we came, which is Luke. Notice he's at the top of our visual stack.

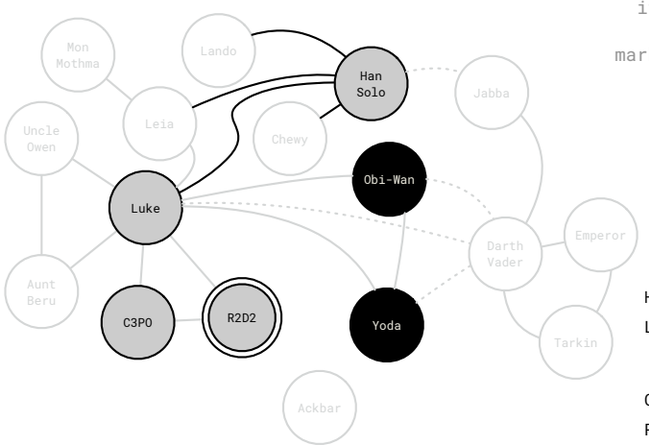
The next node in Luke's list is Han.

And since this example is getting long, and I've shown all the interesting stuff already, I'm just going to burn through the rest of this. If you get the slides, you can click through it to watch how the algorithm progresses.



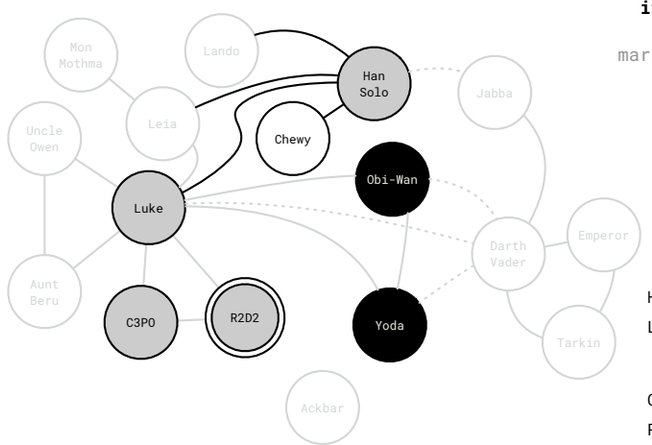

```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



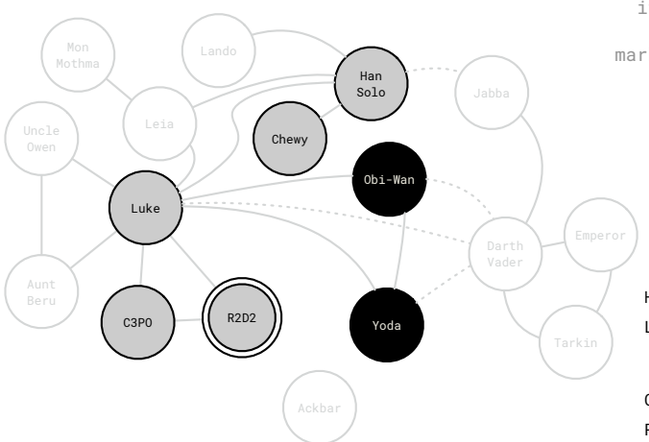

```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Han Solo:	Chewy, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



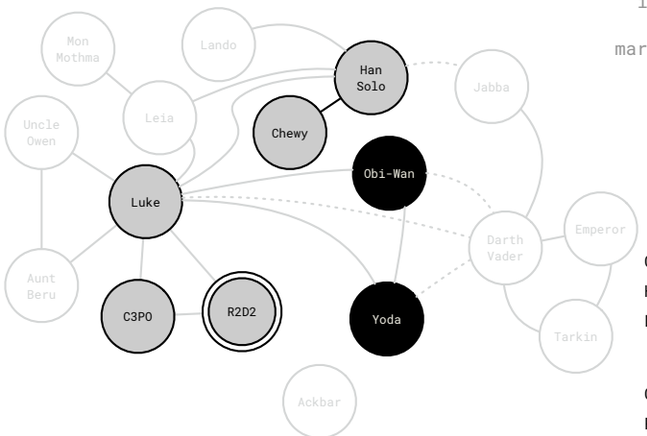

```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Han Solo:	**Chewy**, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



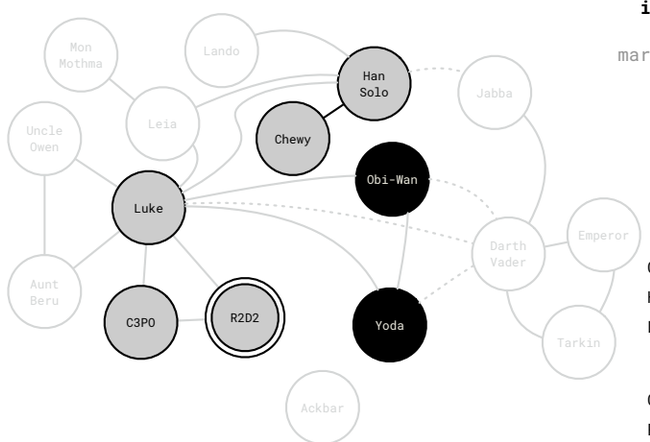

```
DFS(chewy):
  mark chewy gray
  visit chewy
  for each edge e related to chewy:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark chewy black
```

Han Solo:	**Chewy**, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



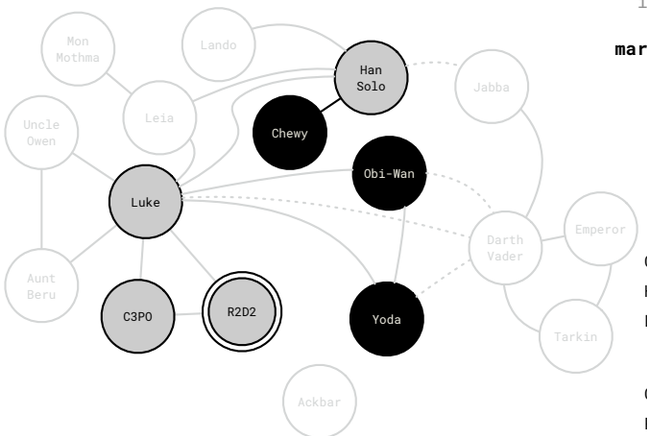

```
DFS(chewy):
  mark chewy gray
  visit chewy
  for each edge e related to chewy:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark chewy black
```

Chewy:	Han Solo
Han Solo:	**Chewy**, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



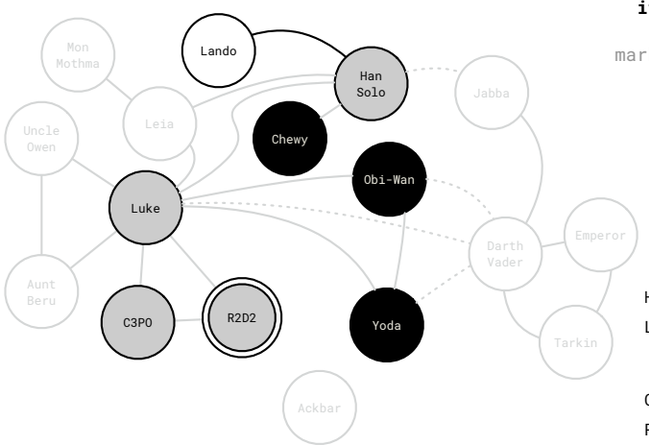

```
DFS(chewy):
  mark chewy gray
  visit chewy
  for each edge e related to chewy:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark chewy black
```

Chewy:	**Han Solo**
Han Solo:	**Chewy**, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



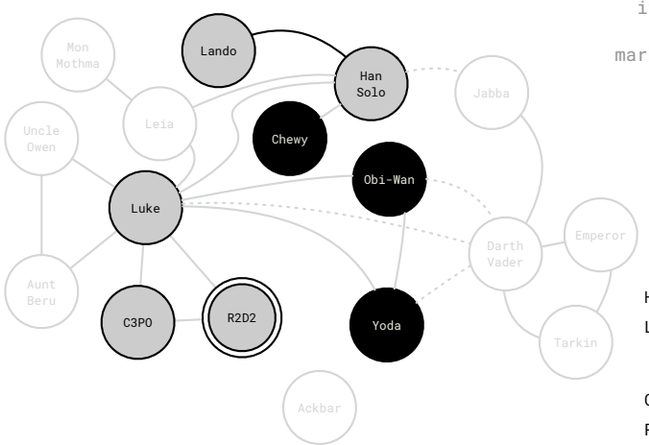

```
DFS(chewy):
  mark chewy gray
  visit chewy
  for each edge e related to chewy:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark chewy black
```

Chewy:	~~Han Solo~~
Han Solo:	**Chewy**, Lando, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



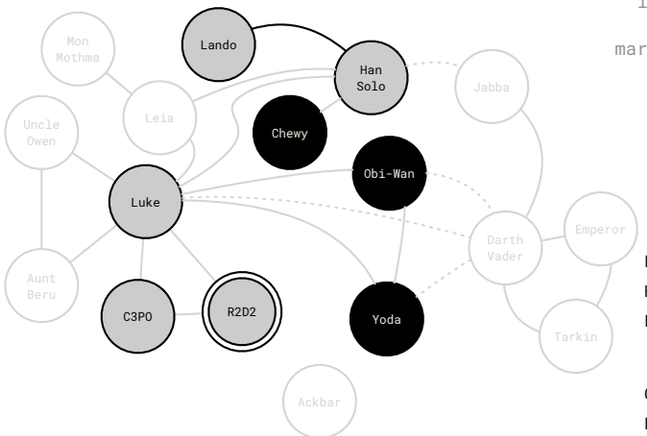

```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Han Solo:	~~Chewy,~~ **Lando**, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



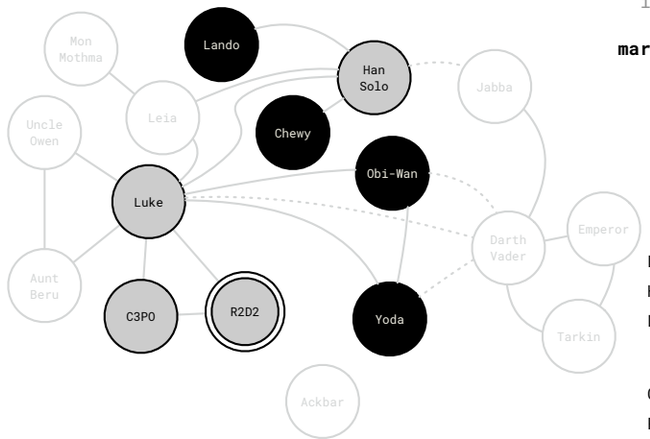

```
DFS(lando):
  mark lando gray
  visit lando
  for each edge e related to lando:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark lando black
```

Han Solo:	~~Chewy,~~ **Lando**, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



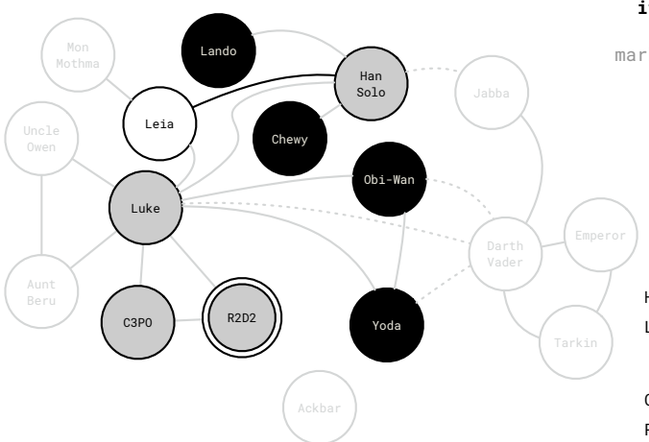

```
DFS(lando):
  mark lando gray
  visit lando
  for each edge e related to lando:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark lando black
```

Lando:	Han Solo
Han Solo:	~~Chewy,~~ **Lando**, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



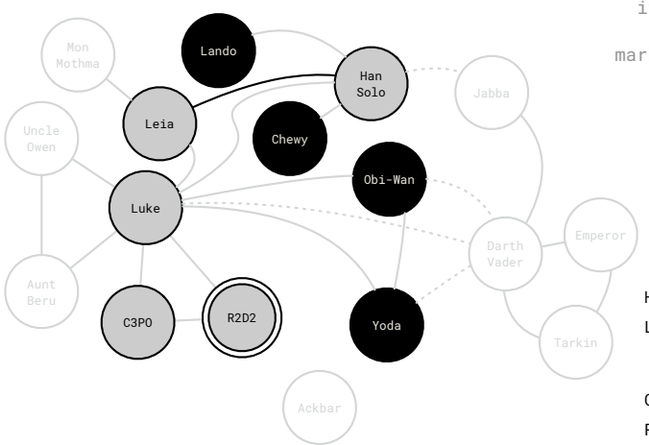

```
DFS(lando):
  mark lando gray
  visit lando
  for each edge e related to lando:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark lando black
```

Lando:	~~Han Solo~~
Han Solo:	~~Chewy,~~ **Lando**, Leia
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



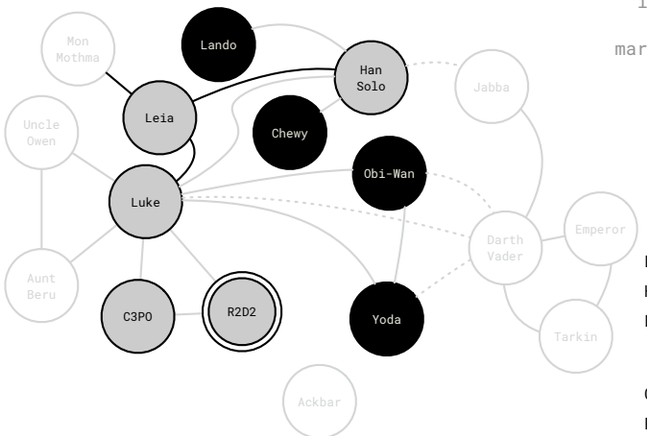

```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Han Solo:	~~Chewy,~~ ~~Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



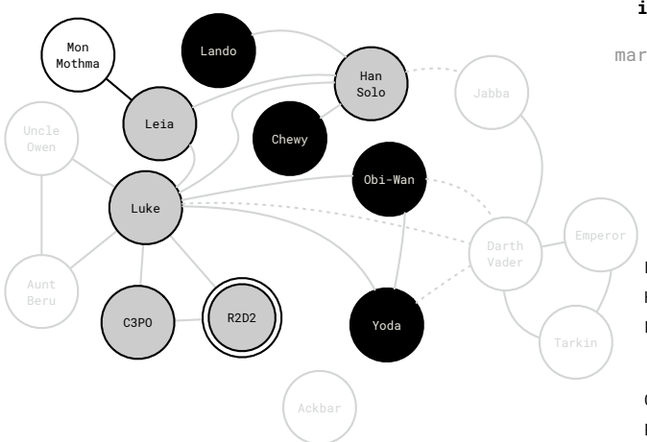

```
DFS(leia):
  mark leia gray
  visit leia
  for each edge e related to leia:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark leia black
```

Han Solo:	~~Chewy,~~ ~~Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



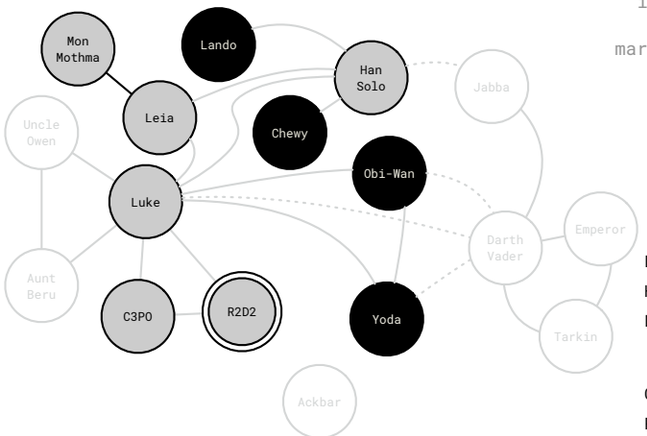

```
DFS(leia):
  mark leia gray
  visit leia
  for each edge e related to leia:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark leia black
```

Leia:	Mon Mothma, Han Solo, Luke
Han Solo:	~~Chewy,~~ ~~Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



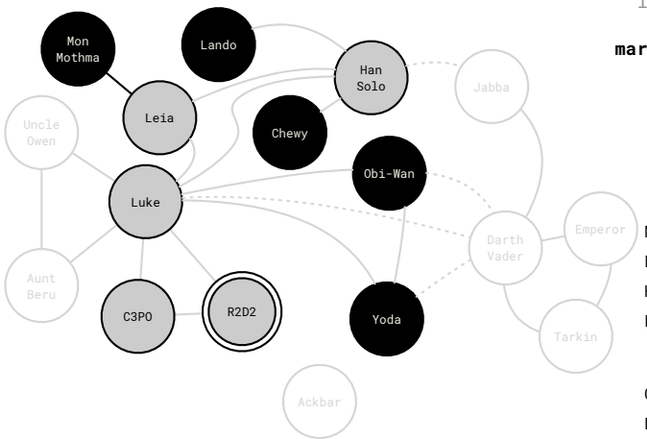

```
DFS(leia):
  mark leia gray
  visit leia
  for each edge e related to leia:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark leia black
```

Leia:	**Mon Mothma**, Han Solo, Luke
Han Solo:	~~Chewy,~~ ~~Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke




```
DFS(mothma):
  mark mothma gray
  visit mothma
  for each edge e related to mothma:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark mothma black
```

Leia:	**Mon Mothma**, Han Solo, Luke
Han Solo:	~~Chewy,~~ ~~Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke




```
DFS(mothma):
  mark mothma gray
  visit mothma
  for each edge e related to mothma:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark mothma black
```

Mon Mothma: ~~Leia~~

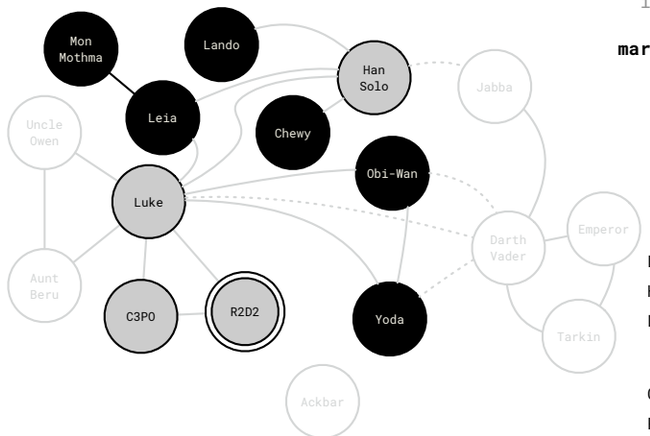
Leia: **Mon Mothma**, Han Solo, Luke

Han Solo: ~~Chewy,~~ ~~Lando,~~ **Leia**

Luke: ~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda

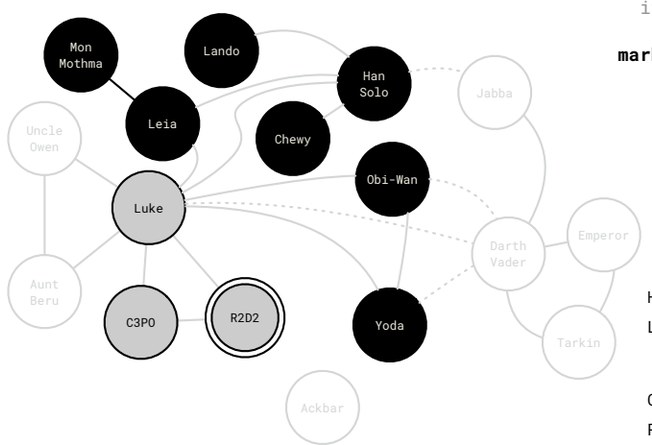
C3PO: ~~R2D2,~~ **Luke**

R2D2: **C3PO**, Luke



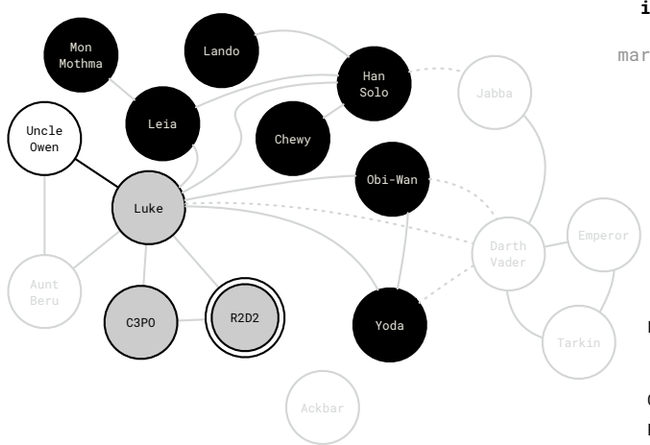

```
DFS(leia):
  mark leia gray
  visit leia
  for each edge e related to leia:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark leia black
```

Leia:	~~Mon Mothma, Han Solo, Luke~~
Han Solo:	~~Chewy, Lando,~~ **Leia**
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke




```
DFS(han):
  mark han gray
  visit han
  for each edge e related to han:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark han black
```

Han Solo:	~~Chewy,~~ ~~Lando,~~ ~~Leia~~
Luke:	~~Obi-Wan,~~ **Han Solo,** Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke

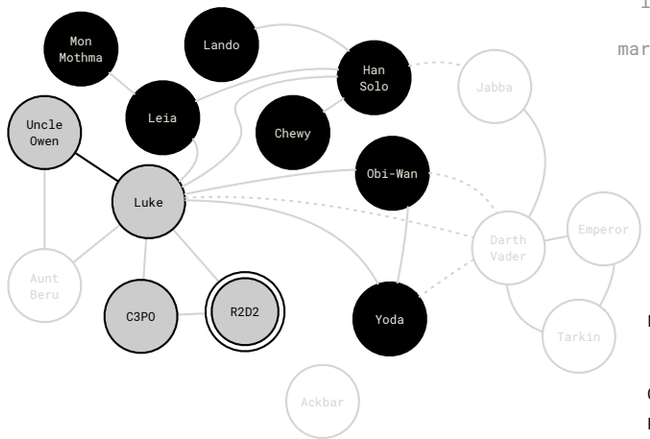



```
DFS(luke):
  mark luke gray
  visit luke
  for each edge e related to luke:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark luke black
```

Luke: ~~Obi-Wan,~~ ~~Han Solo,~~ ~~Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda

C3PO: ~~R2D2,~~ **Luke**

R2D2: **C3PO**, Luke

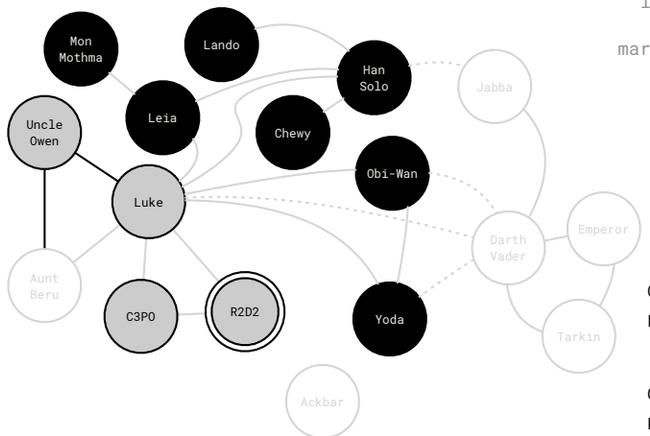



```
DFS(owen):
  mark owen gray
  visit owen
  for each edge e related to owen:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark owen black
```

Luke: ~~Obi-Wan,~~ ~~Han Solo,~~ ~~Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda

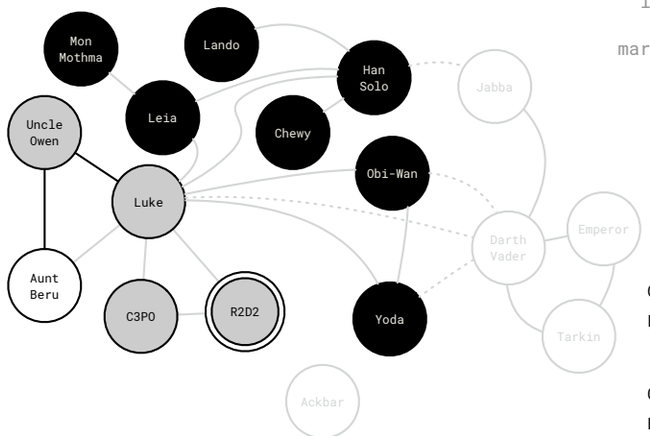
C3PO: ~~R2D2,~~ **Luke**

R2D2: **C3PO**, Luke



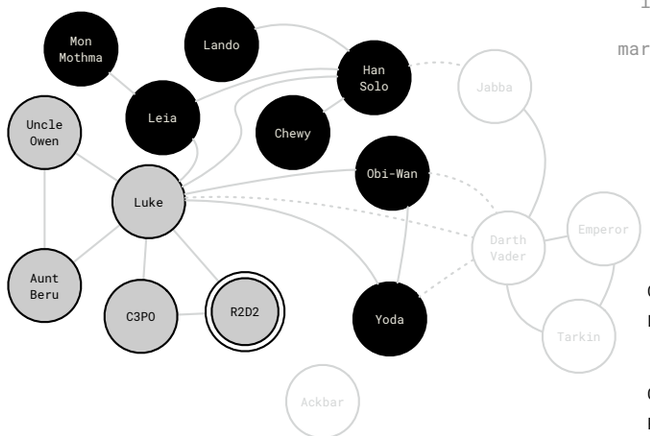

```
DFS(owen):
  mark owen gray
  visit owen
  for each edge e related to owen:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark owen black
```

Owen:	Luke, Aunt Beru
Luke:	~~Obi-Wan,~~ ~~Han Solo,~~ ~~Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



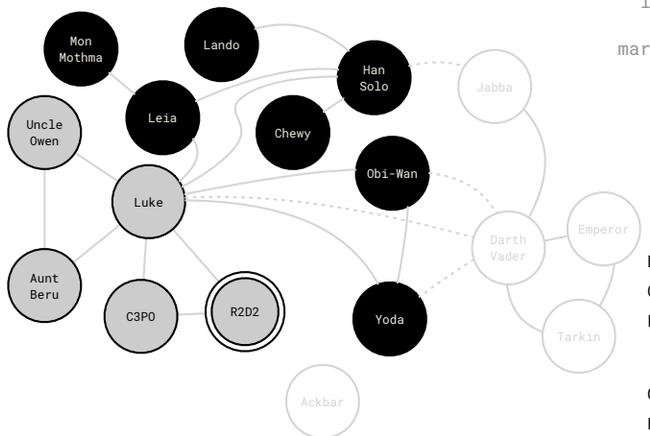

```
DFS(owen):
  mark owen gray
  visit owen
  for each edge e related to owen:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark owen black
```

Owen:	~~Luke~~, **Aunt Beru**
Luke:	~~Obi-Wan, Han Solo, Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



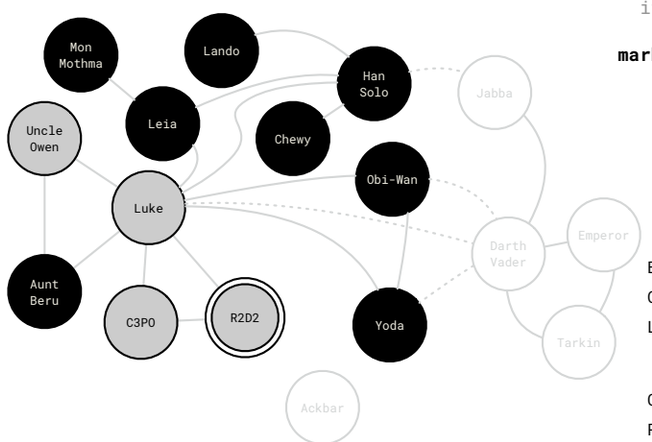

```
DFS(beru):
  mark beru gray
  visit beru
  for each edge e related to beru:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark beru black
```

Owen:	~~Luke~~, **Aunt Beru**
Luke:	~~Obi-Wan, Han Solo, Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



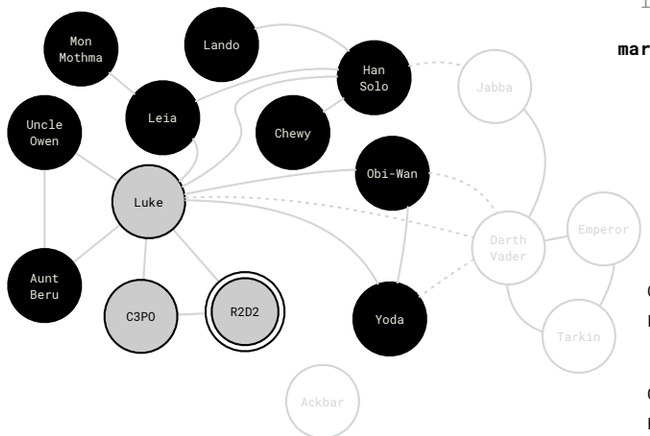

```
DFS(beru):
  mark beru gray
  visit beru
  for each edge e related to beru:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark beru black
```

Beru:	Luke, Uncle Owen
Owen:	~~Luke~~, **Aunt Beru**
Luke:	~~Obi-Wan, Han Solo, Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke



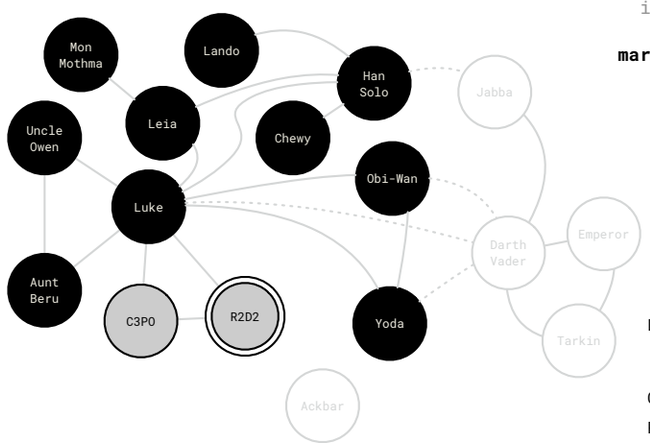

```
DFS(beru):
  mark beru gray
  visit beru
  for each edge e related to beru:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark beru black
```

Beru:	~~Luke, Uncle Owen~~
Owen:	~~Luke~~, **Aunt Beru**
Luke:	~~Obi-Wan, Han Solo, Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke




```
DFS(owen):
  mark owen gray
  visit owen
  for each edge e related to owen:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark owen black
```

Owen:	~~Luke, Aunt Beru~~
Luke:	~~Obi-Wan, Han Solo, Leia,~~ **Uncle Owen**, Aunt Beru, C3PO, R2D2, Yoda
C3PO:	~~R2D2,~~ **Luke**
R2D2:	**C3PO**, Luke

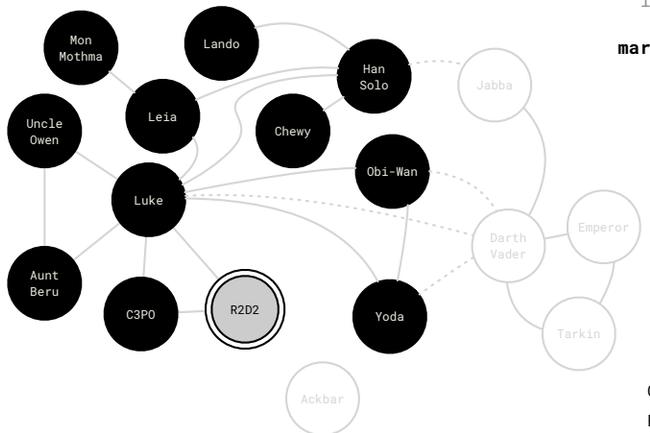



```
DFS(luke):
  mark luke gray
  visit luke
  for each edge e related to luke:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark luke black
```

Luke: ~~Obi-Wan, Han Solo, Leia, Uncle Owen, Aunt Beru, C3PO, R2D2, Yoda~~

C3PO: ~~R2D2,~~ **Luke**

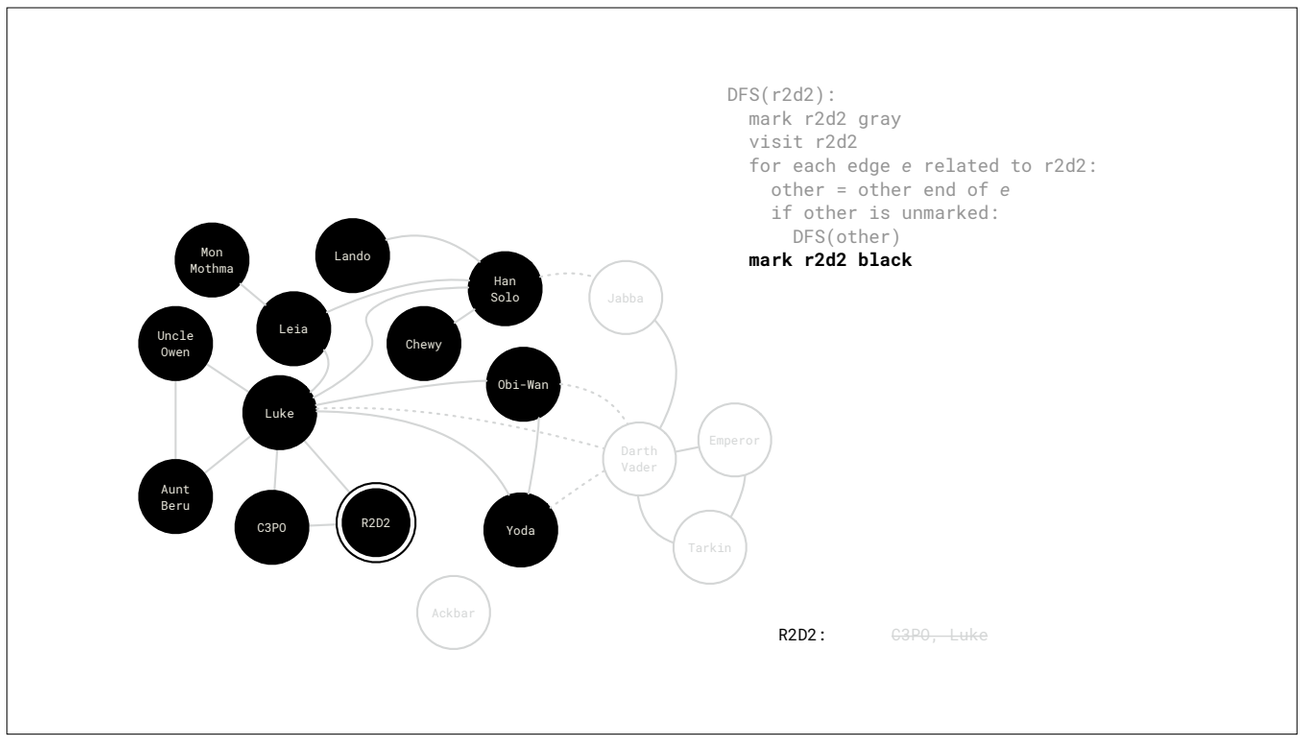
R2D2: **C3PO**, Luke




```
DFS(c3po):
  mark c3po gray
  visit c3po
  for each edge e related to c3po:
    other = other end of e
    if other is unmarked:
      DFS(other)
  mark c3po black
```

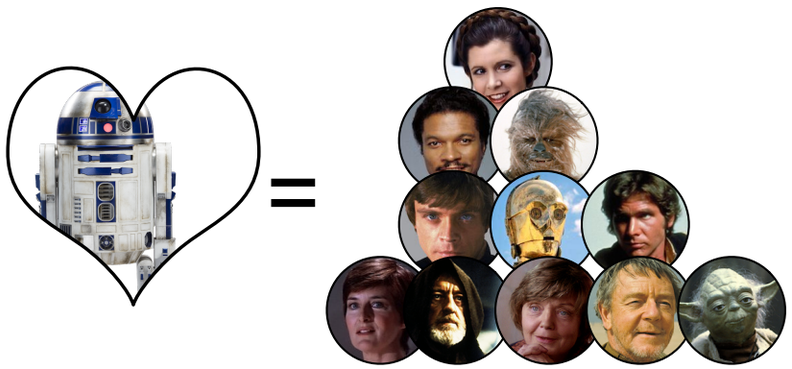
C3PO: ~~R2D2~~, ~~Luke~~

R2D2: **C3PO**, Luke



And at last, we're marking R2D2 as complete.

We set out to find out who in the Star Wars social network were friends-of-friends of R2D2. Using a Depth-first search traversing friendly edges, our solution is the set of all nodes that are now colored Black. This is the entire cast of good guys except for Ackbar, who wasn't connected to anyone.



So, here’s the answer! Brought to you by Depth-first search.

Episode 6

Breadth-First Search (BFS)

Since I'm sure you haven't had enough of new graph algorithms, we're now going to tackle Breadth-First Search on the same Star Wars social network.

```
BFS(start_node):
    queue = empty queue
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        visit node
        optionally, mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                add other_node to queue
                mark other_node gray
```

The names of these algorithms are really quite apt. With a depth-first search, you go as deep as you can and eventually get around to finishing where you started. But with breadth-first search, nodes that are close to the start node are completed before nodes that are farther out. This is the basic, vanilla breadth-first search. **<next build>**

First thing you'll see is that we're not doing this recursively. Instead of letting our recursive stack of calls implicitly keep track of things, we're going to a data structure to keep track of what's up next. **<next>** That's what the empty queue is for on the first line.

Breadth-first search is all about popping the first element from the queue, adding all of it's unmarked neighbors to the queue. For that to work, we need to populate the queue. **<next>** So on the second line we add our starting node to the queue.

We're marking nodes as 'in progress' by coloring them gray. This is used to make sure we only add a node to the queue one time. **<next>** That's why on line three we mark our starting node gray. **<next>** Then we enter a big loop, removing the first item from the queue, and then visiting it.

<next> Then we can optionally mark the node as black, which indicates we're finished. It helps the visualization, so I'm going to do it, but just be aware that it isn't strictly necessary for the algorithm to work.

<next> Now, for all the edges related to our current node, take a look at all of them and if they're not yet marked, then treat them just like we treated the start node. **<next>** Add it to the queue and color it gray. And that's really all there is to it.

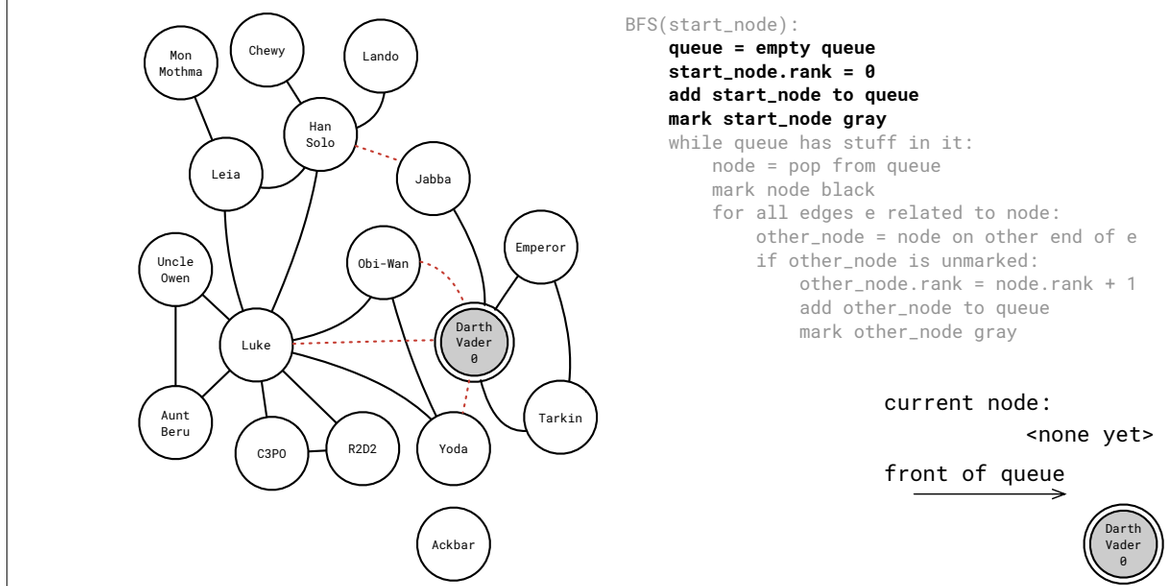
How Far To Darth?

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

We're going to use a slight modification of the vanilla BFS to calculate the 'rank' of each node connected to Darth Vader. This will tell us the smallest number of hops it will take to get to Darth.

We'll modify the basic BFS by adding a 'rank' field to each node. Before adding the node to the queue, we set it's rank to one plus the rank of whatever node we're looking at right now.

<next> The difference is just in the first initialization part, we set the start node to have rank zero. And then later inside the loop, whenever we add a node to the queue, we set it's rank to be one plus it's parent.




```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

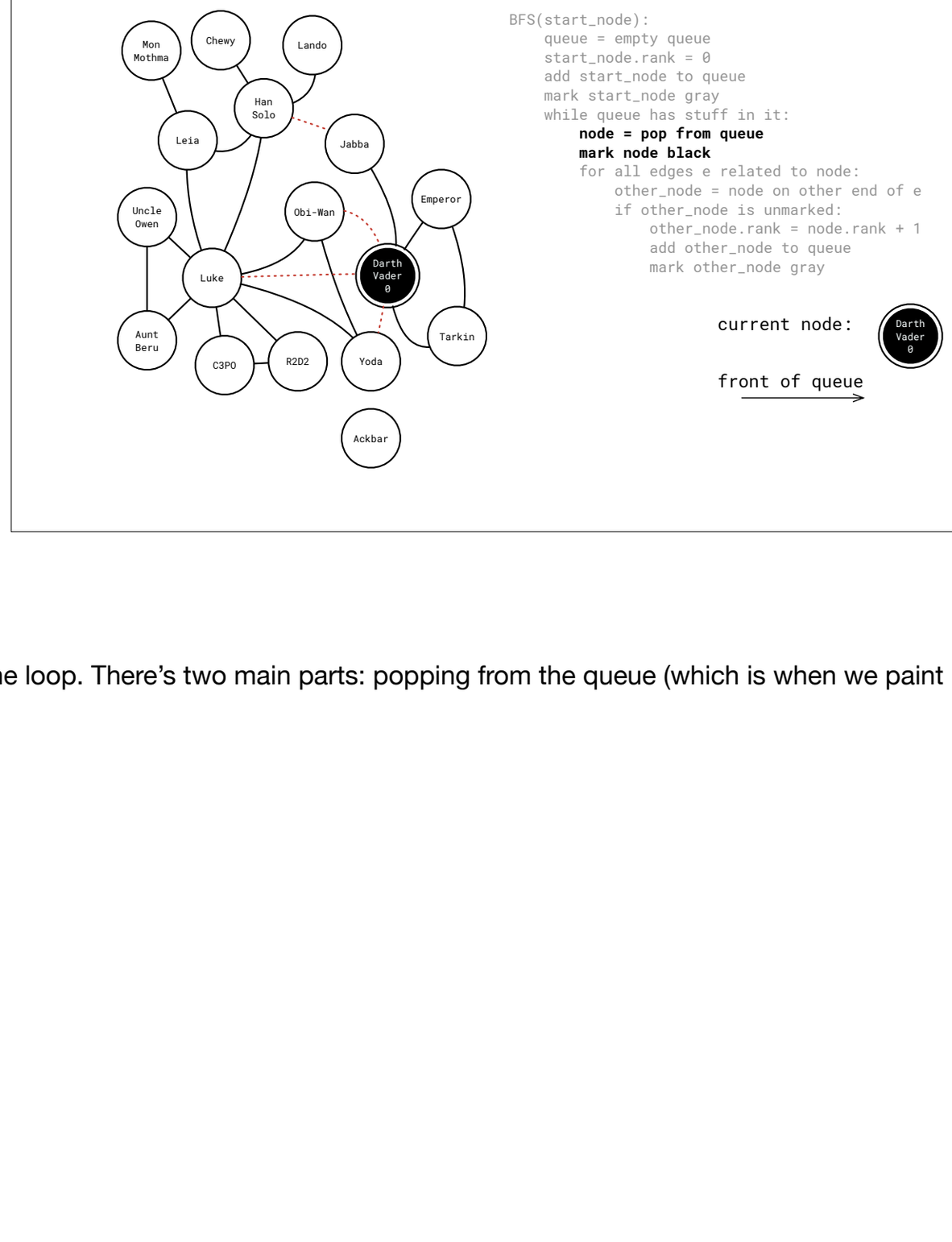
current node:

<none yet>

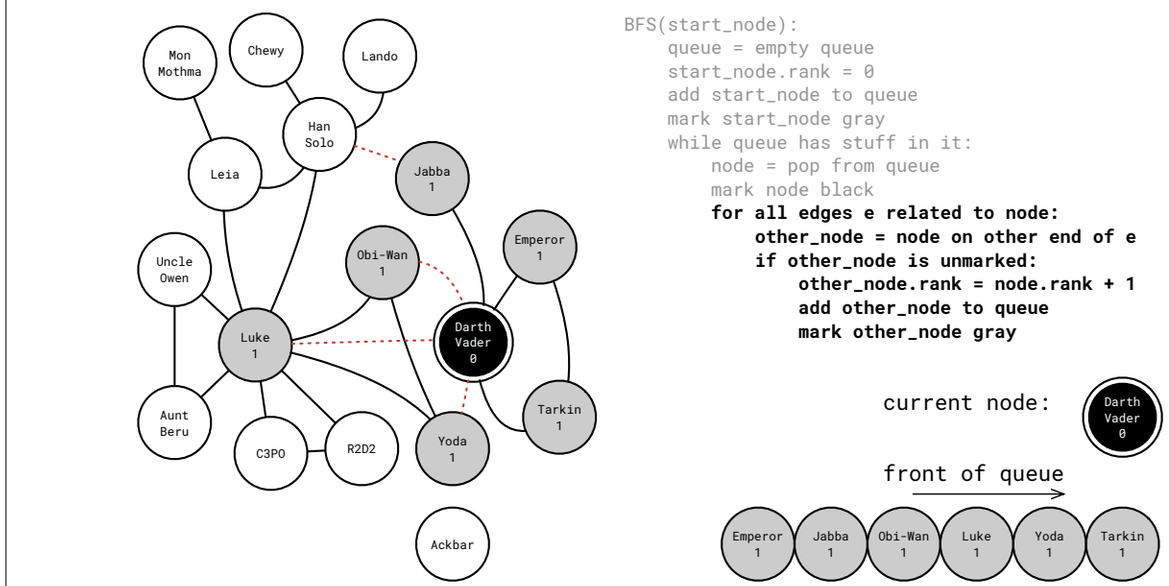
front of queue →

OK, let's get started. We're ignoring edge types in this graph, so we'll follow hostile and friendly edges alike.

First thing we do is establish a queue with just our gray-painted starting node in it. I'll put it down there at the bottom, and the front of the queue is to the right.



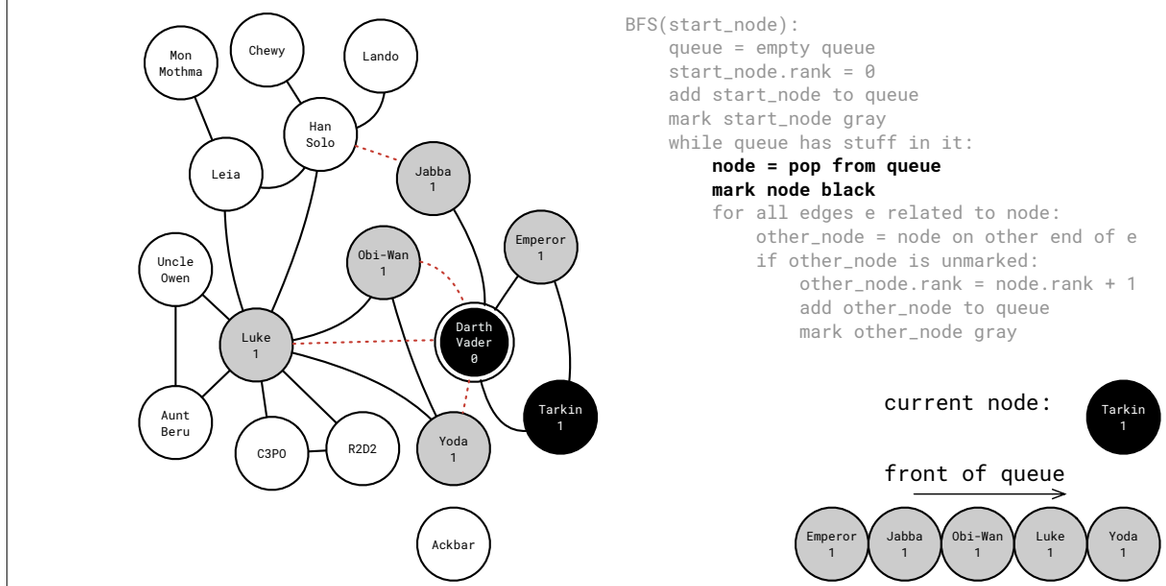
We spend basically all the time inside the loop. There’s two main parts: popping from the queue (which is when we paint nodes black), and enqueing that node’s neighbors that aren’t yet in the queue.




```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```


So we add all of Darth’s adjacent nodes, since none of them were in the queue.

And notice that we gave each one of them a rank of one, which is Darth’s rank (zero), plus one.

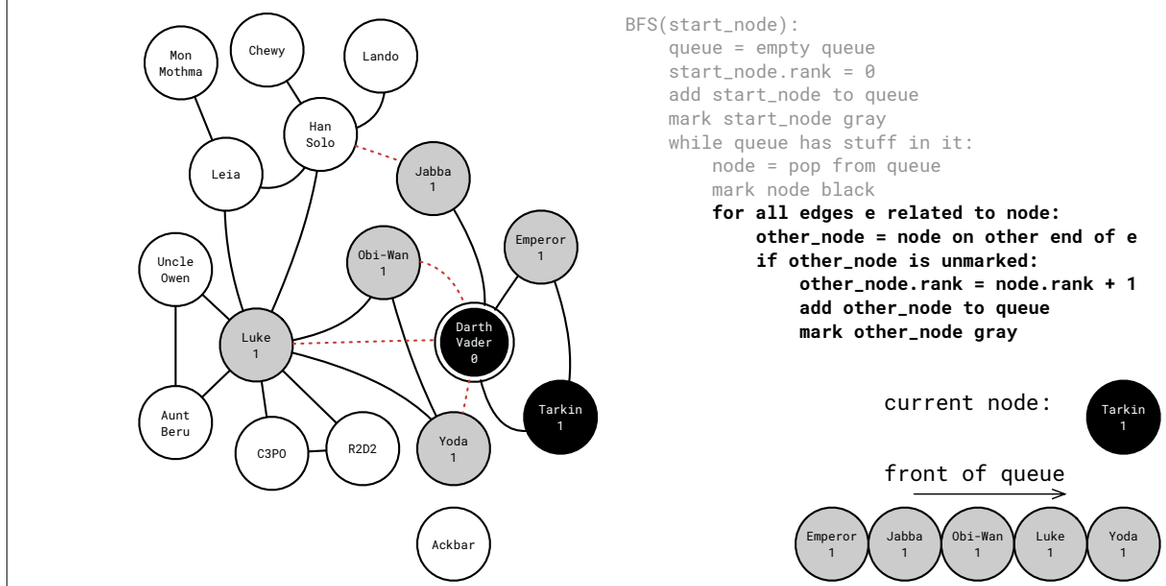



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Tarkin 1

front of queue →

Done with Darth, now we pop the next node in the queue, which happens to be Tarkin. Color it black.

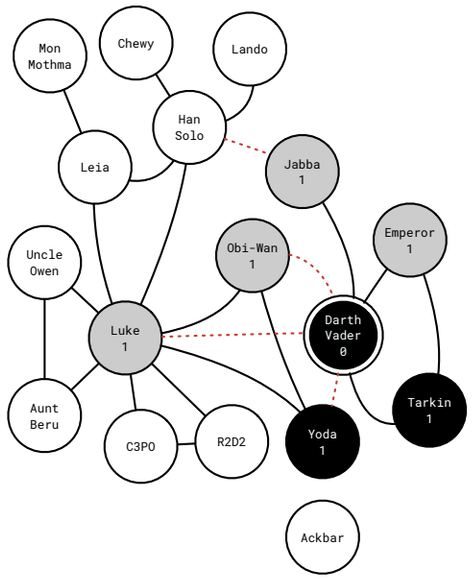



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Tarkin 1

front of queue →

Tarkin doesn’t have any adjacent nodes that aren’t already in the queue, so nothing interesting happens here.

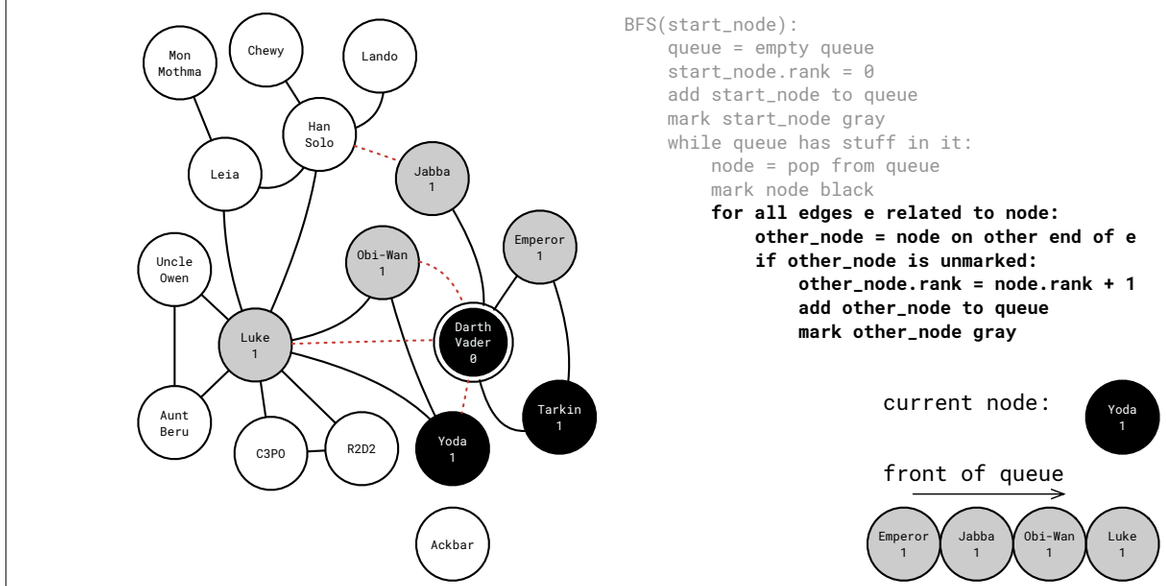



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Yoda 1

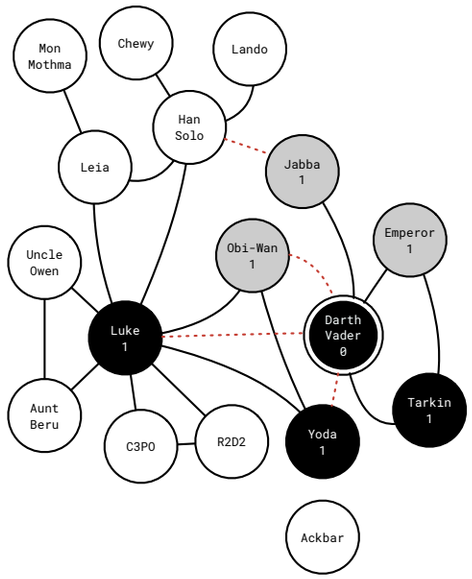
front of queue →

Pop Yoda, mark him done.




```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```


None of Yoda’s adjacent nodes are available, so again nothing is added to the queue.

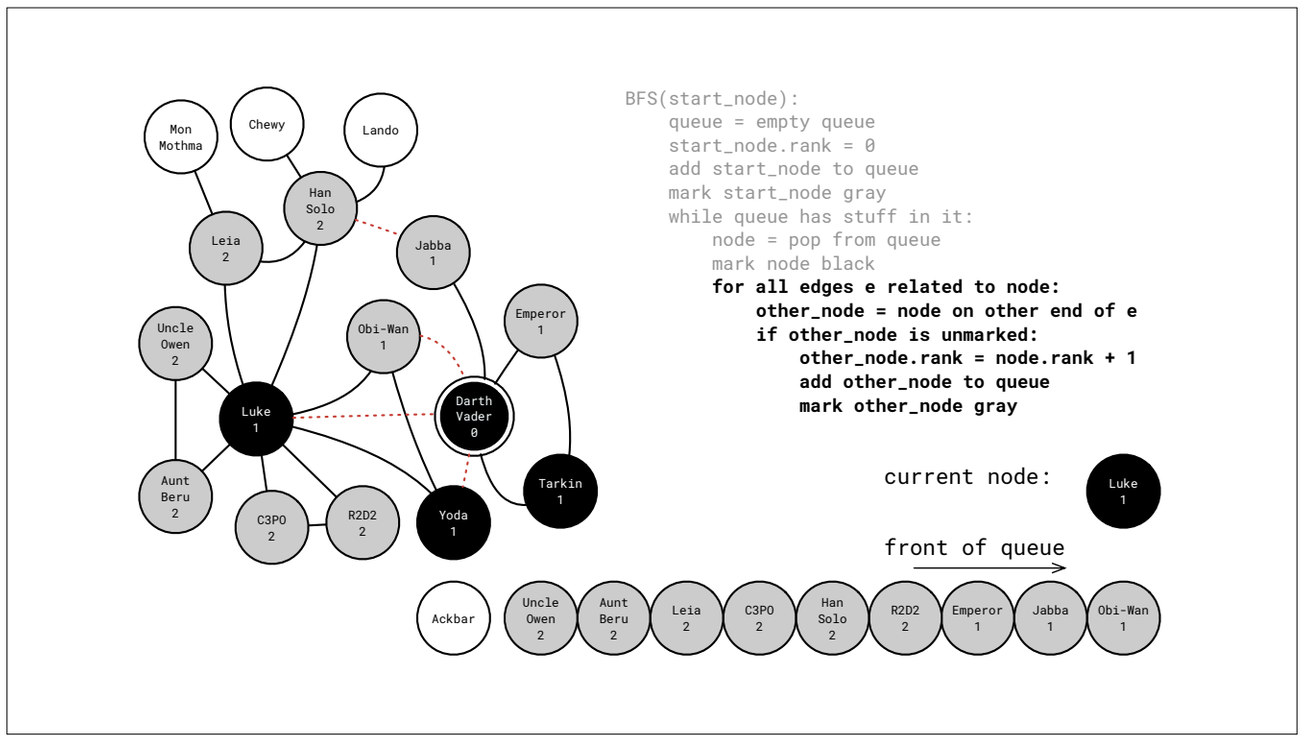



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

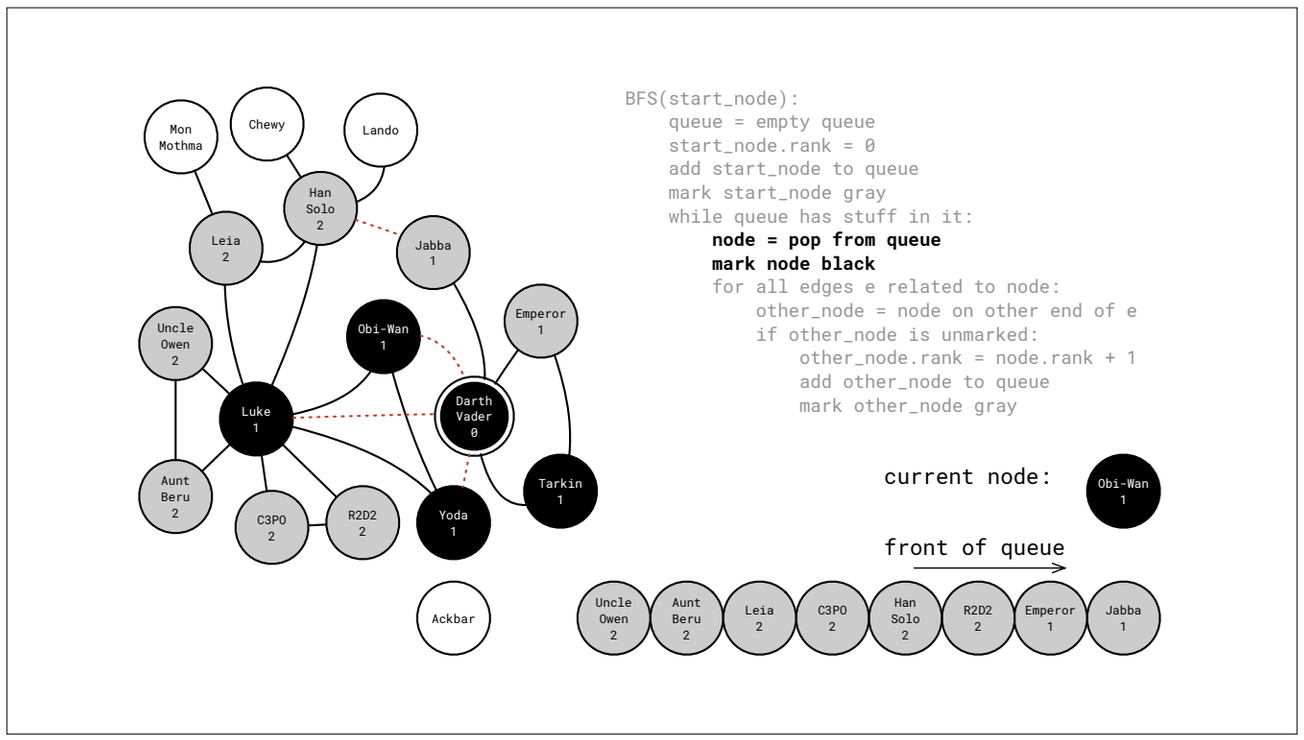
current node: Luke 1

front of queue →

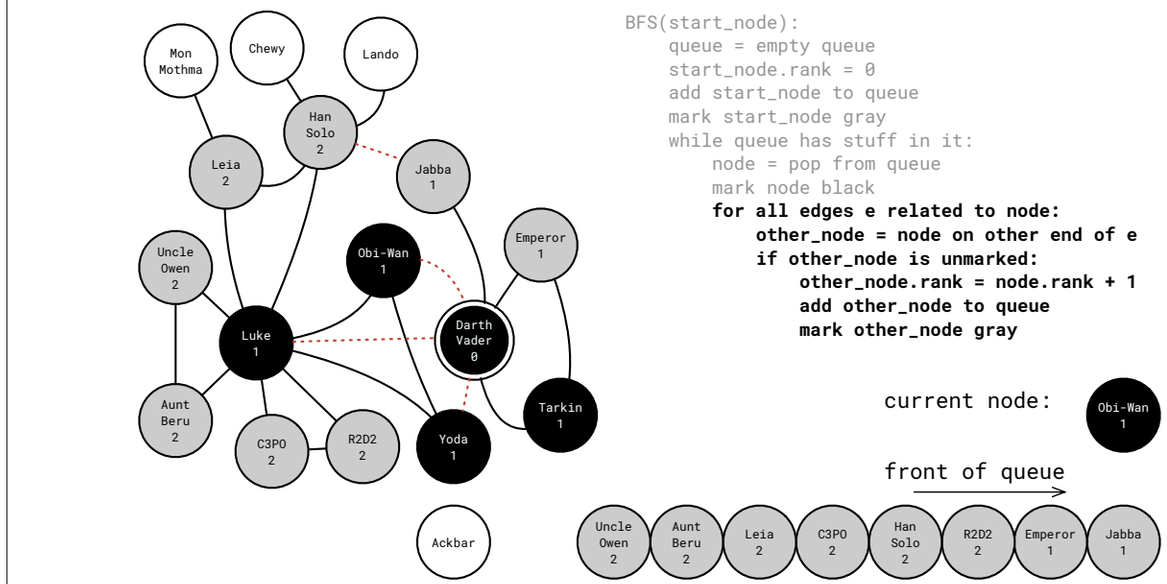
Pop Luke, mark him done.



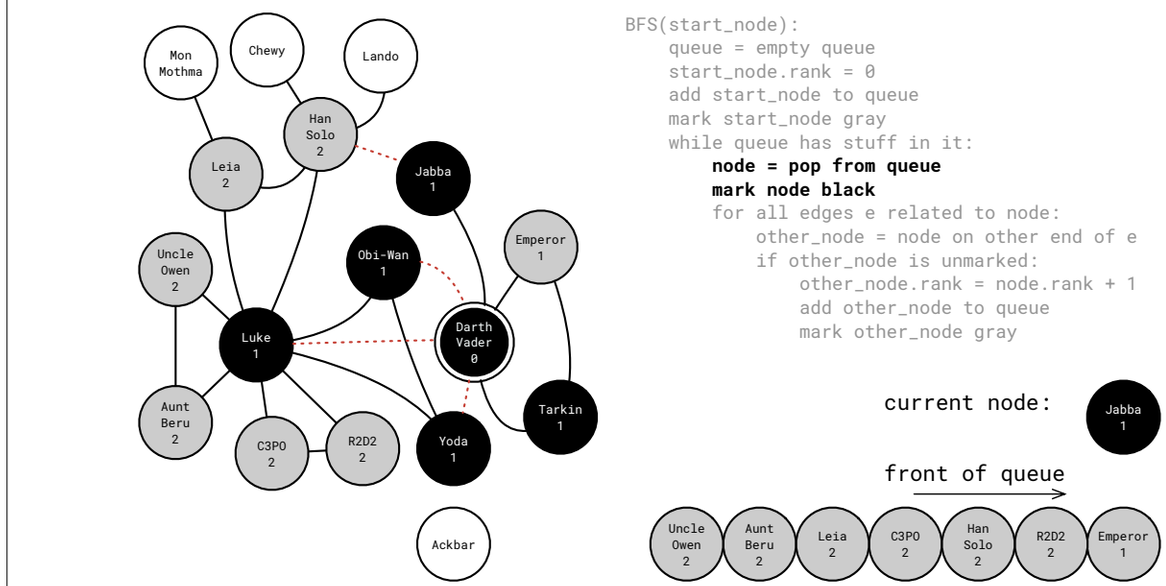
Enqueue Luke’s 500+ connections. Of all of the adjacent nodes to Luke, only Obi-Wan was already in the queue, and Darth was already marked completed so we don’t enqueue them. Notice that all of these newly enqueued nodes have a rank of 2 now.


Mon Mothma
Chewy
Lando
Han Solo 2
Leia 2
Jabba 1
Uncle Owen 2
Obi-Wan 1
Emperor 1
Luke 1
Darth Vader 0
Aunt Beru 2
C3PO 2
R2D2 2
Yoda 1
Tarkin 1
Ackbar
BFS(start_node):
queue = empty queue
start_node.rank = 0
add start_node to queue
mark start_node gray
while queue has stuff in it:
node = pop from queue
mark node black
for all edges e related to node:
other_node = node on other end of e
if other_node is unmarked:
other_node.rank = node.rank + 1
add other_node to queue
mark other_node gray
current node:
Obi-Wan 1
front of queue
Uncle Owen 2
Aunt Beru 2
Leia 2
C3PO 2
Han Solo 2
R2D2 2
Emperor 1
Jabba 1

Pop Obi-Wan…



Enqueue nothing because they’re already in the queue.

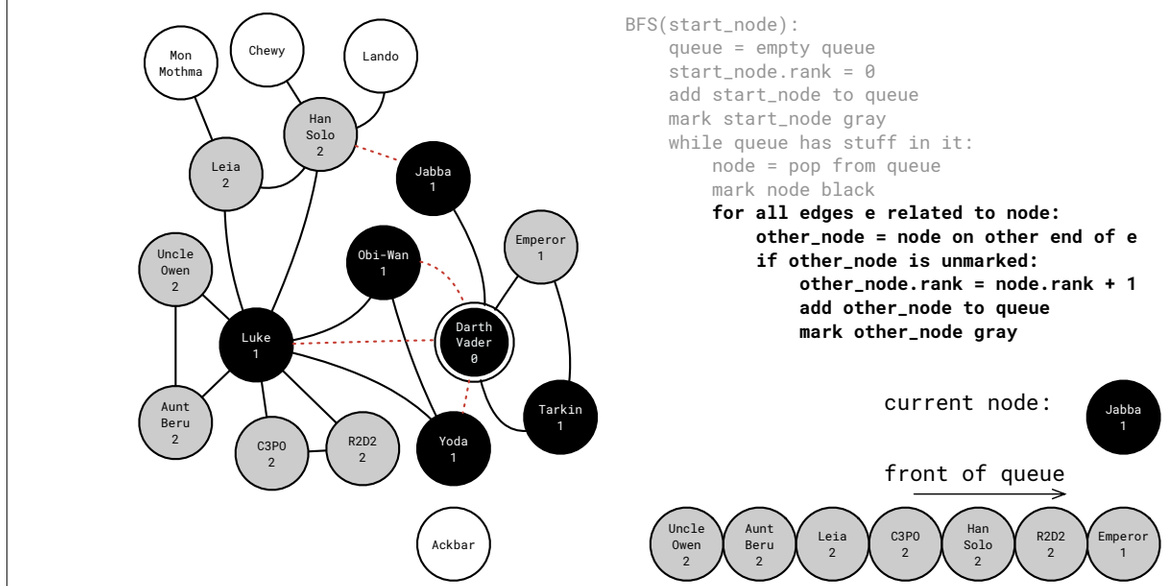



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Jabba 1

front of queue →

Pop Jabba…



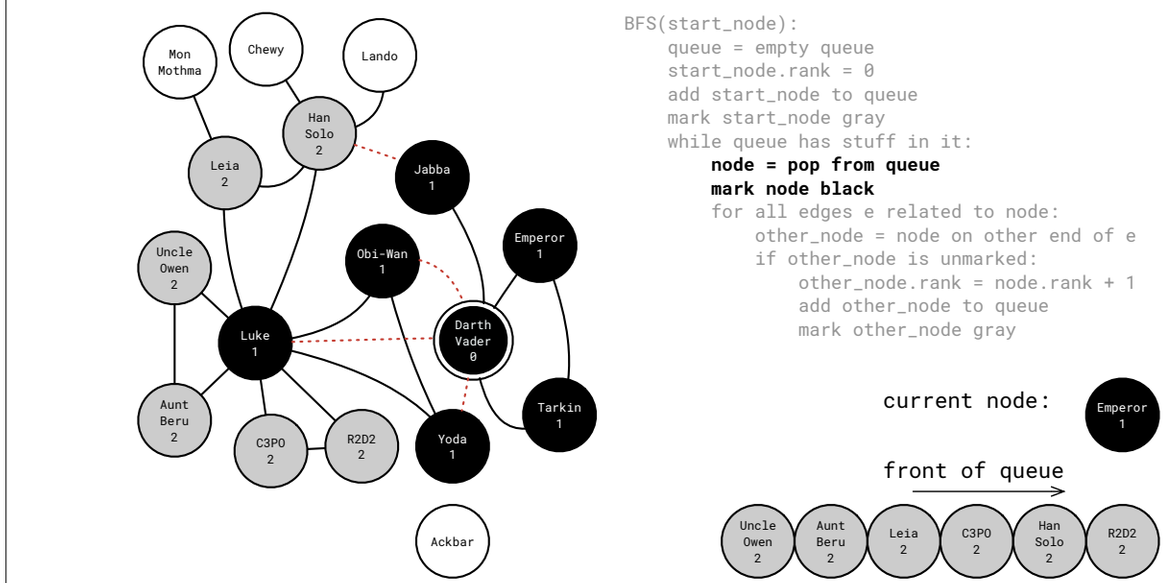

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Jabba 1

front of queue →

Uncle Owen 2 | Aunt Beru 2 | Leia 2 | C3PO 2 | Han Solo 2 | R2D2 2 | Emperor 1

Again nothing to add…

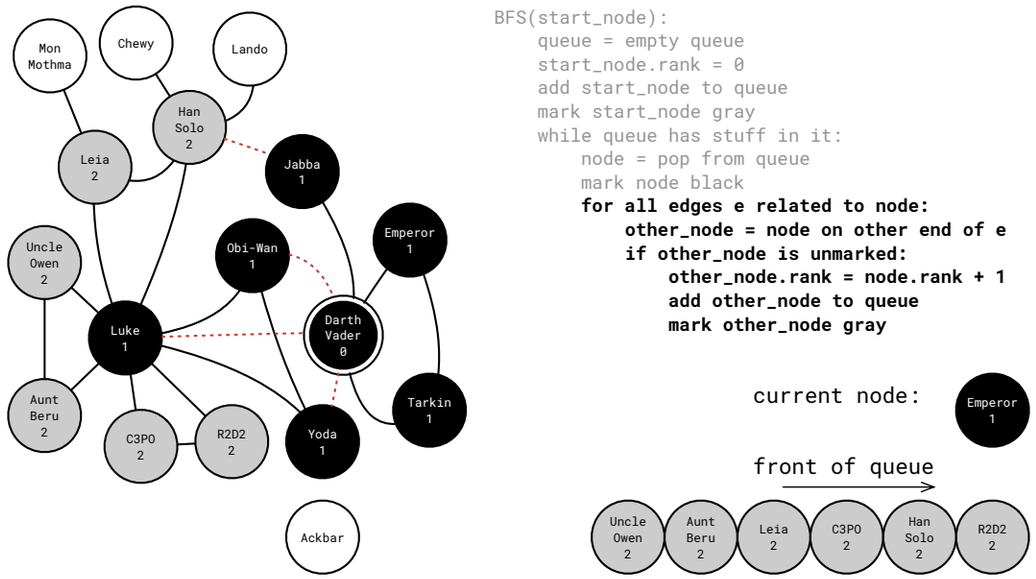



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Emperor 1

front of queue →

Pop the Emperor…

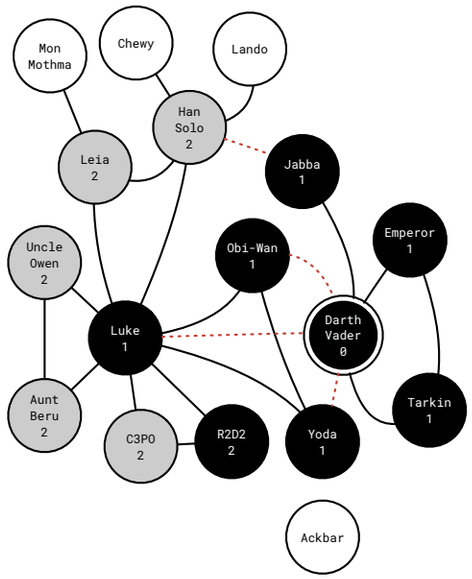



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Emperor 1

front of queue →

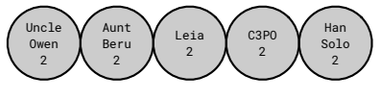
Nothing to add…



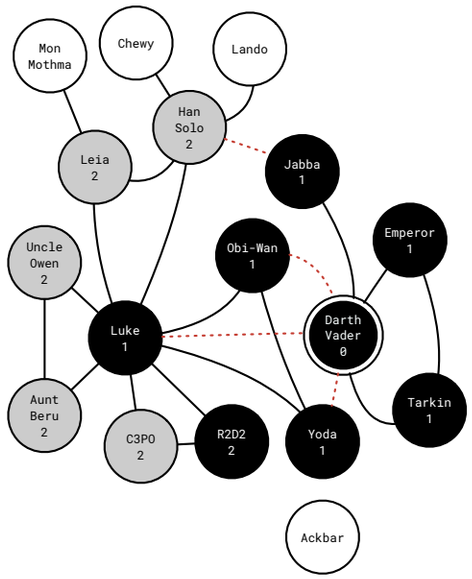

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: R2D2 2

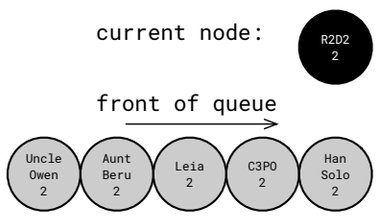
front of queue →



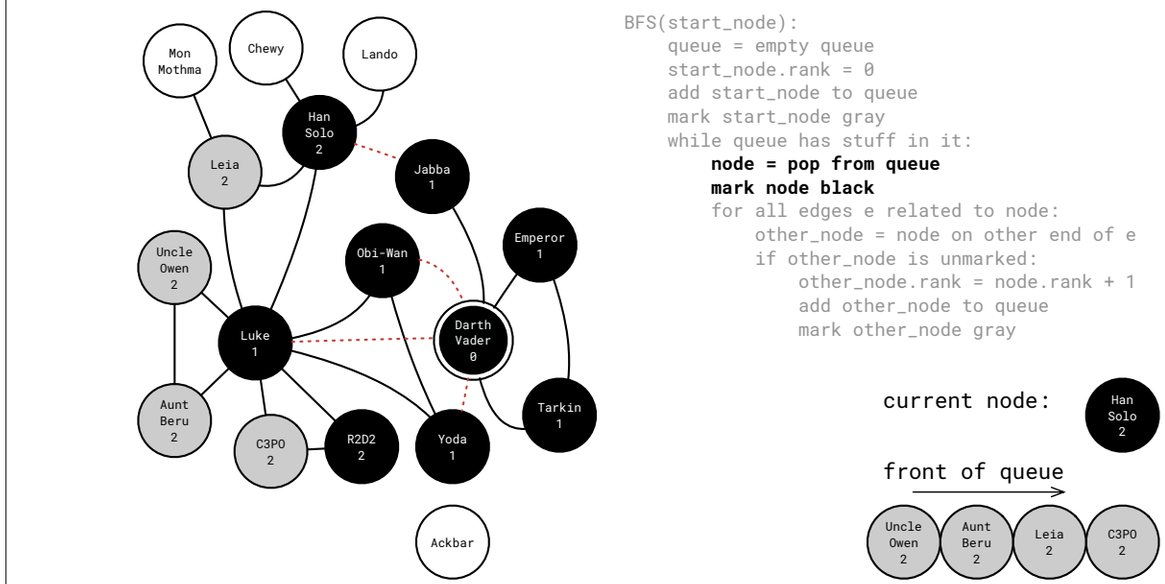
Pop R2D2…




```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```



Nothing to add… Boring!!



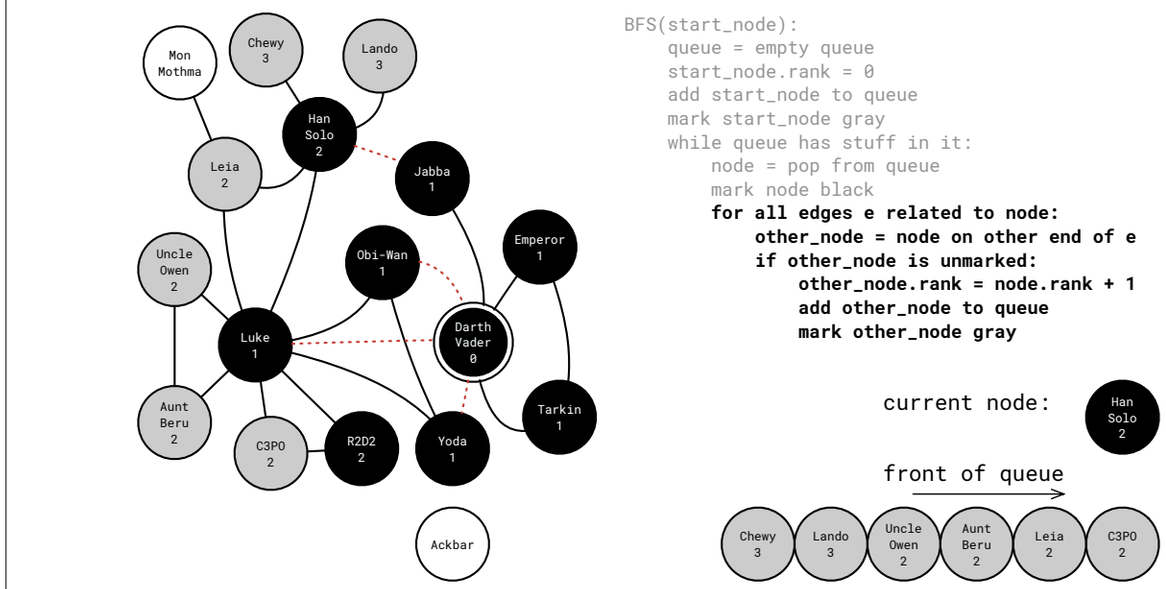

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Han Solo 2

front of queue →

Uncle Owen 2 | Aunt Beru 2 | Leia 2 | C3PO 2

Pop Han Solo.



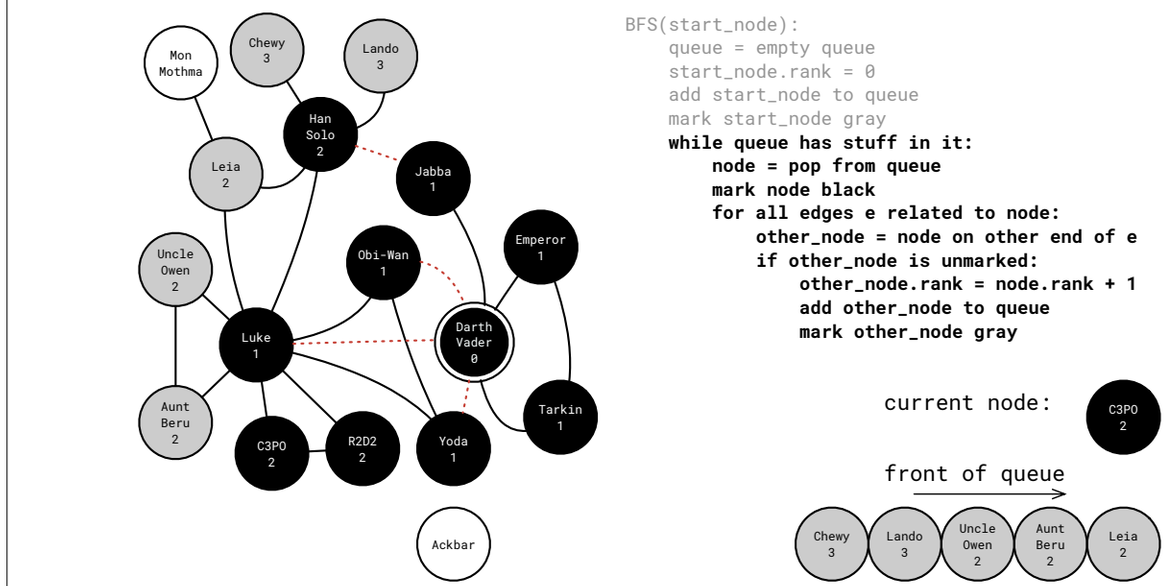

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Han Solo 2

front of queue →

Chewy 3 | Lando 3 | Uncle Owen 2 | Aunt Beru 2 | Leia 2 | C3PO 2

Finally, more nodes to enqueue. Chewy and Lando get rank 3.



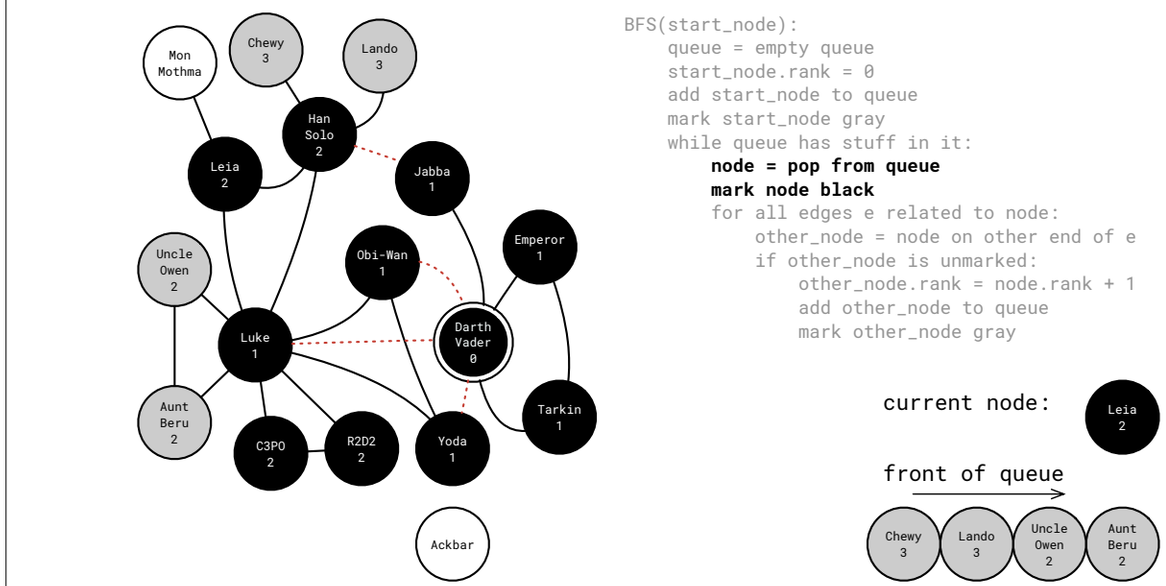

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: C3PO 2

front of queue →

Chewy 3 | Lando 3 | Uncle Owen 2 | Aunt Beru 2 | Leia 2

C3PO pops and has no neighbors to enqueue.



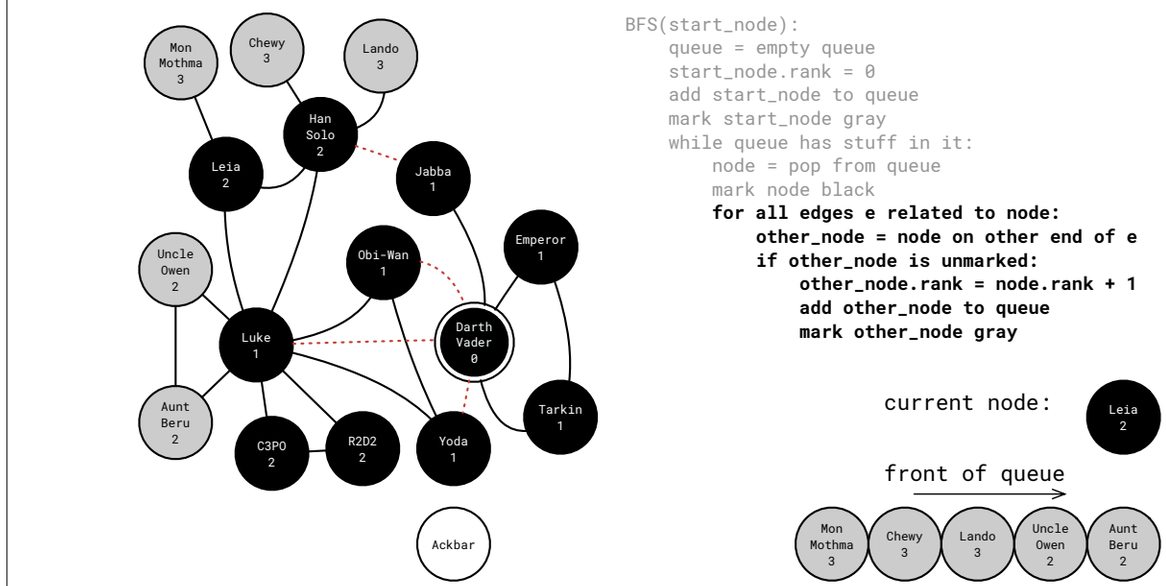

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Leia 2

front of queue →

Chewy 3 | Lando 3 | Uncle Owen 2 | Aunt Beru 2

Leia pops, but she has an unvisited neighbor.



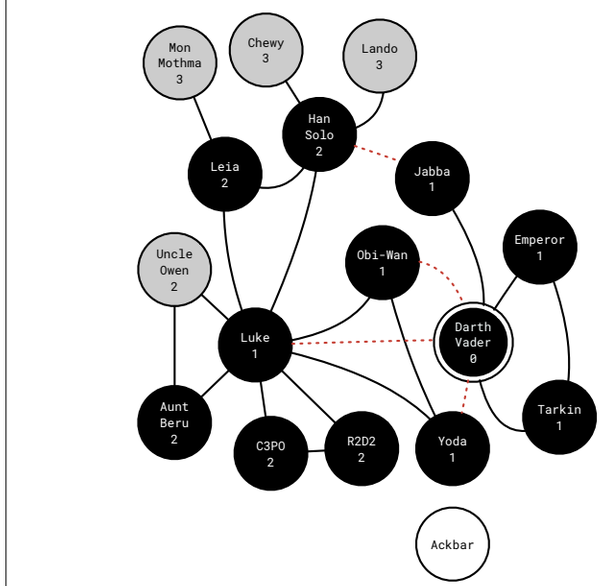

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Leia 2

front of queue →

Mon Mothma 3 | Chewy 3 | Lando 3 | Uncle Owen 2 | Aunt Beru 2

So we add Mon Mothma to the queue, also with rank 3. Remember, the rank enqueued nodes get is one plus the rank of the node we're visiting when we add them.



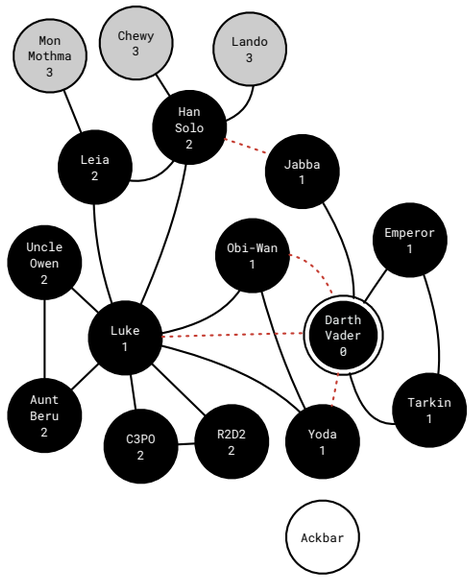

```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Aunt Beru 2

front of queue →

Mon Mothma 3 | Chewy 3 | Lando 3 | Uncle Owen 2

And the rest of the queue…

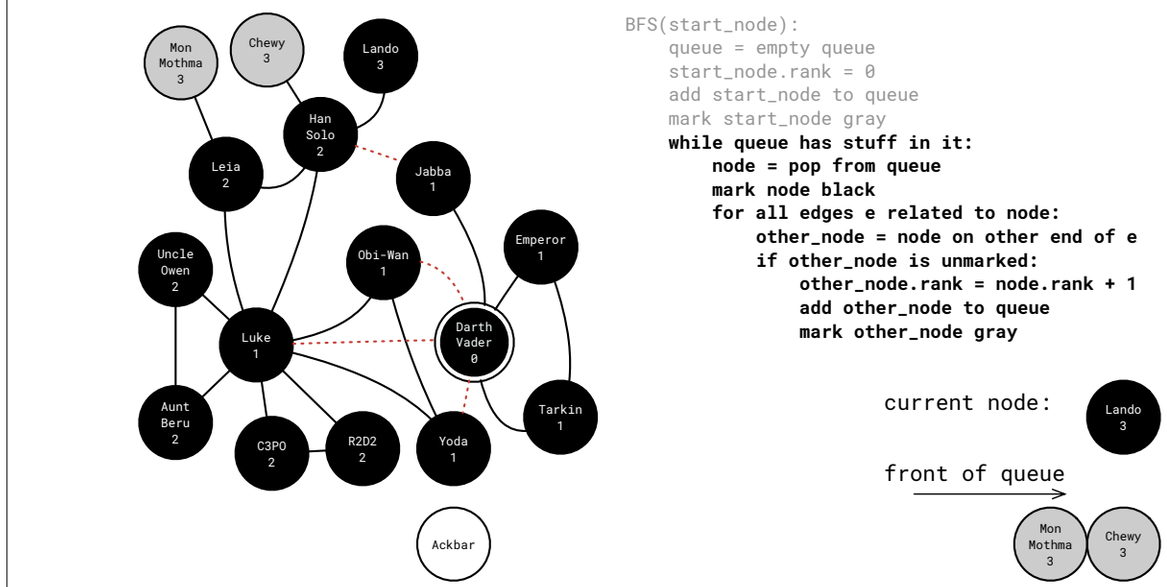



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Uncle Owen 2

front of queue →

Just pops…

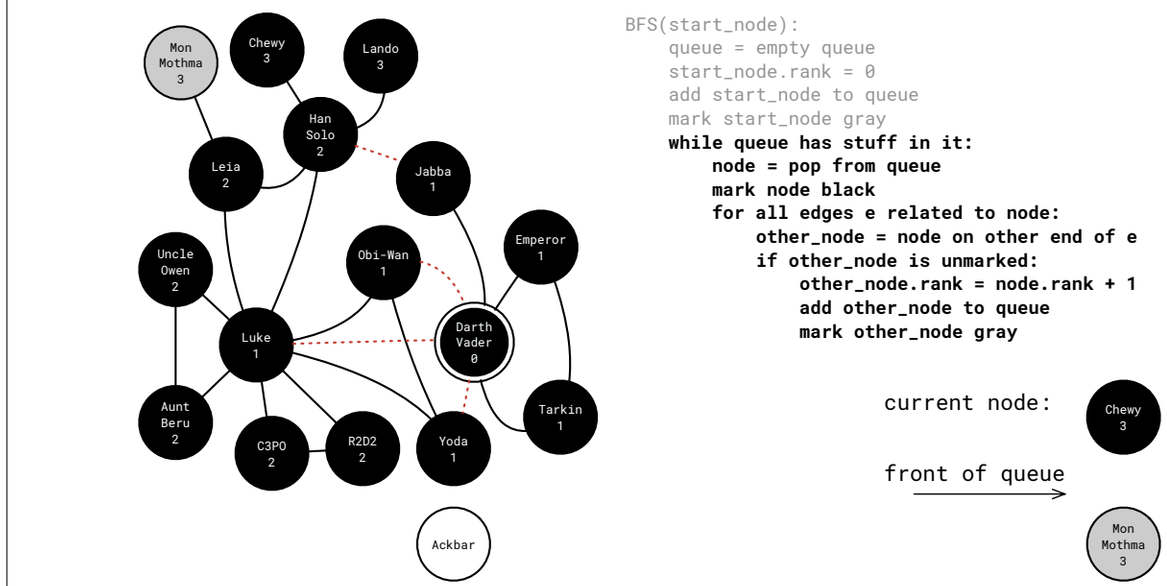



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Lando 3

front of queue → Mon Mothma 3, Chewy 3

and is generally boring.

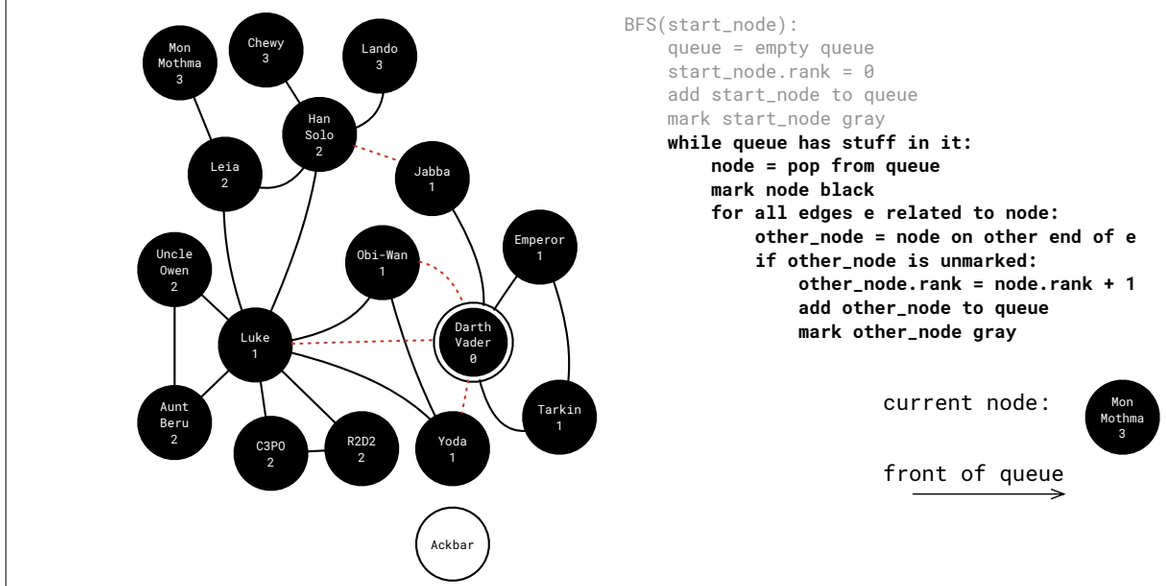



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Chewy 3

front of queue → Mon Mothma 3

Because there’s nothing left to enqueue.

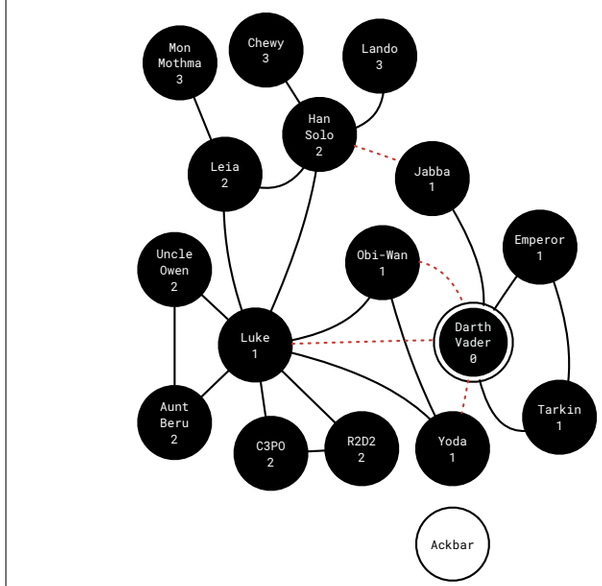



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node: Mon Mothma 3

front of queue →

Except for Admiral Ackbar. But he’s not connected to anyone, so that’s sad.

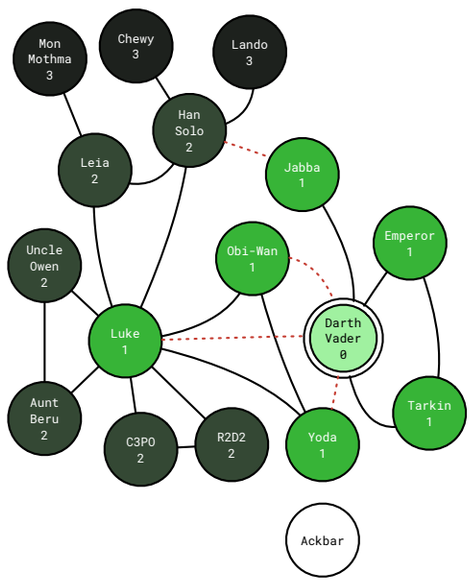



```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node:
<now empty>

front of queue →

And now our queue is empty, which is the condition we need to stop looping. So we're done!




```
BFS(start_node):
    queue = empty queue
    start_node.rank = 0
    add start_node to queue
    mark start_node gray
    while queue has stuff in it:
        node = pop from queue
        mark node black
        for all edges e related to node:
            other_node = node on other end of e
            if other_node is unmarked:
                other_node.rank = node.rank + 1
                add other_node to queue
                mark other_node gray
```

current node:

<now empty>

front of queue →

If we look at the ranks of all nodes, and color code them thusly, it is easier to see ranks.

And that was a basic breadth-first search!

Episode 7

Directed Acyclic Graphs (DAGs)

This episode gives several examples of a specific but common kind of graph called a directed acyclic graph, which are super useful and super common, so we usually just call them DAGs.

We saw from our binary tree days that you can add constraints on a data structure to get useful behaviors, like enforcing a sort order on a binary tree to give us a binary search tree, and enforcing a node coloring scheme to give us balanced red-black trees.

Well, we can add constraints to a plain old graph to give us a DAG. So let's check them out!

DAGs are Everywhere

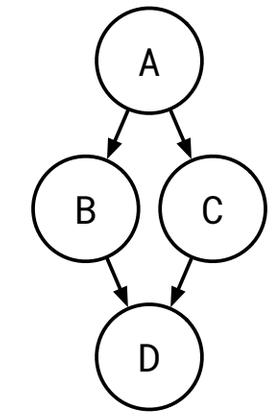
- **Directed Edges (arrows)**
- **No Cycles Allowed**

DAGs are _everywhere_.

They are directed graphs that don't allow cycles. If you start from any given node, you won't be able to follow edges and get back to where you started.

For example!

Simple DAG



✓ right

No matter where we start, node A, B, whatever, we can never get back to where we start if we obey they arrow directions. Eventually we end up stuck in a terminal node like D.

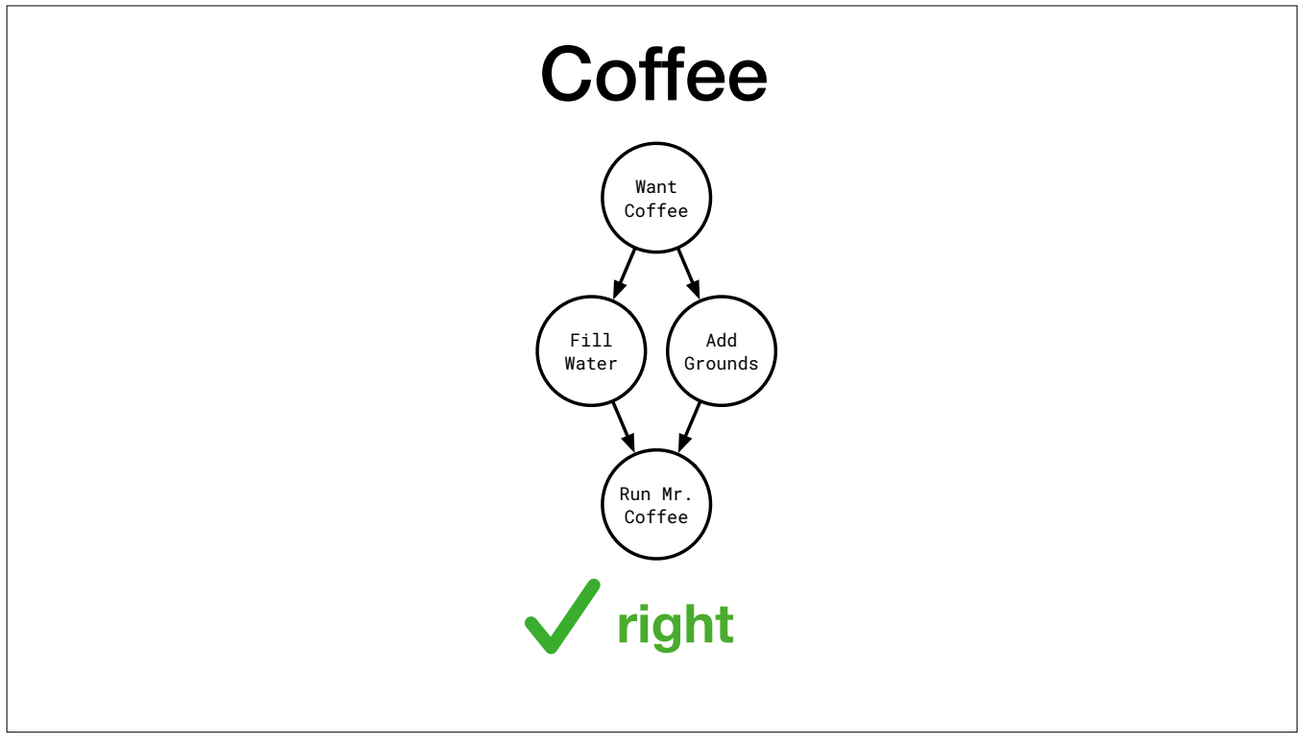
Why Do We Care?

- **Model Precedence**
- **DAG required for many algorithms**

We'll see a ton of examples after this slide, but to make all of that make more sense... why do we care about DAGs?

Well, they model precedence nicely. In our homework assignments, we've been using Makefiles, which specifies the order to compile and link things. Or if you have a spreadsheet where a cell depends on other cells, which in turn depend on other cells, you can model that as a DAG as well. And if there's a cycle in your spreadsheet, you get that horrible "#REF!" error, and it isn't a DAG.

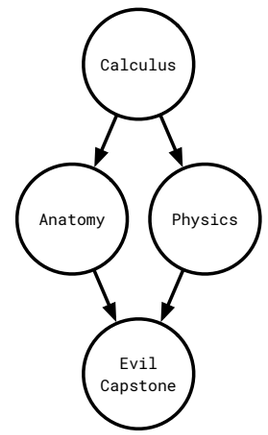
If you have a DAG, you can run certain efficient algorithms on the graph to find shortest or longest paths. But if you don't have a DAG, you have to use less efficient ones, and some tasks like finding the longest path, are intractable.



The nodes could indicate processes that you have to do; the edges represent the order you have to do them.

We'll have to do _all_ the tasks, but it doesn't matter if we fill the water or add the grounds first. It only matters that we do it _before_ we activate Mr. Coffee.

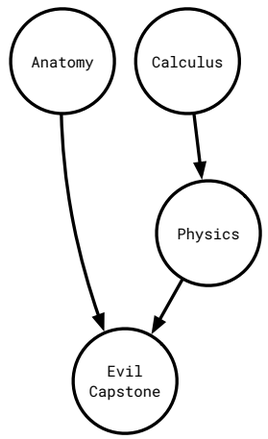
Degree Requirements



✓ right

A Mad Scientist major at the University of Doom has to take classes in a certain order. Calculus is a pre-requisite to both Anatomy and Physics.

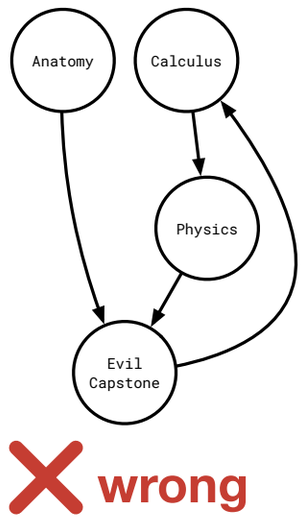
Degree Requirements



✓ right

This would also be a correct DAG. The Department of Mad Science removed the calculus re-req for Anatomy. So now we can take anatomy and calculus at the same time, whereas earlier we had to take Calc first, and _then_ anatomy.

Impossible Degree Requirements



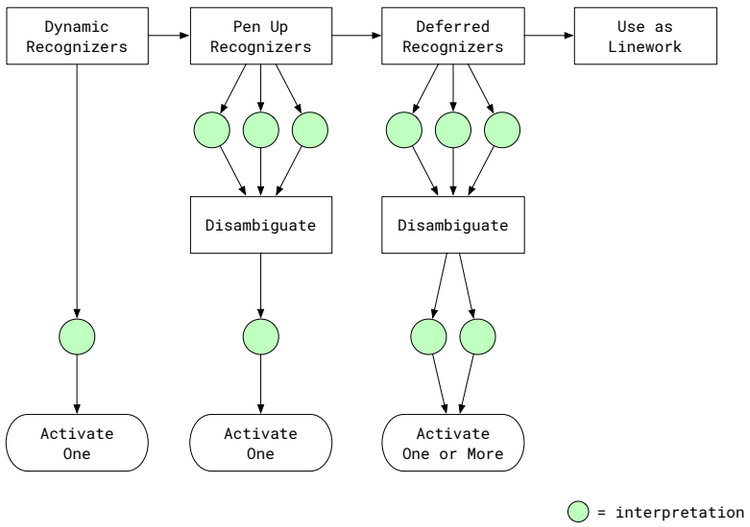
wrong

If the Department changed the pre-req rules to be like this, we would be in school for ever... Or, we wouldn't get started at all, I'm not sure.

Here there's an endless loop that cycles between three classes involving calculus. This is actually pretty close to what it feels like to be an engineering major.

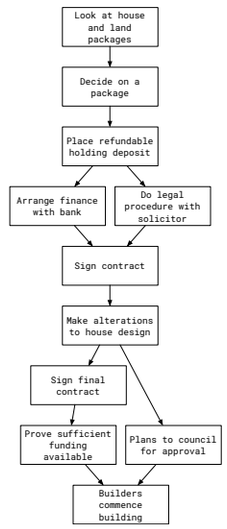
Anyway, this is impossible because the degree requirements can't be modeled as a DAG.

From My Dissertation



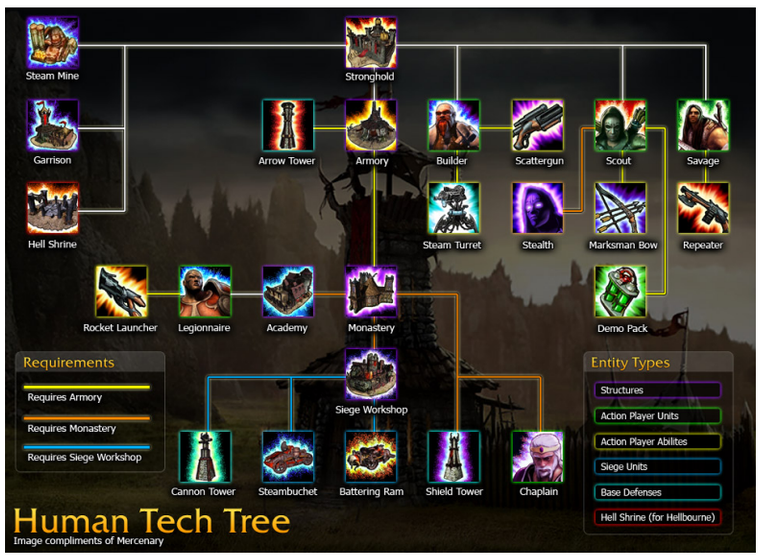
This is an actual slide from my thesis defense some years ago. It models a process for recognizing freehand sketches. In the previous slides we ended up with a single terminal node, like brewing coffee or graduating with a degree in Mad Science. In this diagram there are several terminal nodes, and that's perfectly OK, because there were several ways the sketch could be recognized. It is directed, and there are no cycles, so it's a DAG.

Project Management



Or in project management, you've got a bunch of tasks, some of which are prerequisites to others. Each task has some estimated length of time. Ultimately there's usually a bottom node, and the path that takes us from start to finish with the longest overall length of time is called the 'critical path' which tells us how long the overall project might take.

Video Game Tech Tree



Last example, I swear. Video Games! A Tech Tree is a common concept in games, where if you want to get the advanced things you have to get the basic things in a certain order. They're often but not always DAGs.

In Sum

We can model these things with DAGs:

- **Making Coffee**
- **Degree Requirements**
- **Sketch Recognition**
- **Project Management**
- **Tech Trees**

<< and much, much more >>

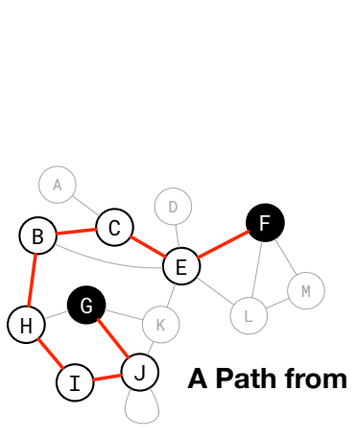
This episode was just a grab-bag of examples of DAGs, so now when you see them in the wild, you'll be able to identify them.

Episode 8

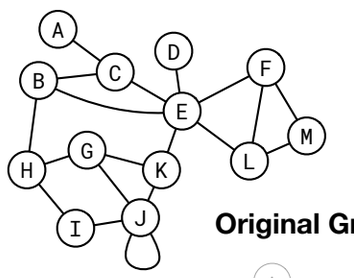
Spanning Trees

There are a bunch of graph algorithms that use this idea called a spanning tree. In the abstract, a spanning tree is any tree that connects all of the nodes of a graph using existing edges. It basically describes one possible ordering or hierarchy that can be imposed on a graph. And as we build a spanning tree, we can also classify edges in a helpful way.

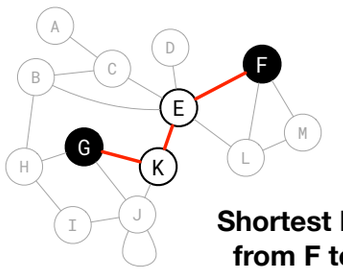
Why We Care



A Path from F to G



Original Graph



Shortest Path
from F to G

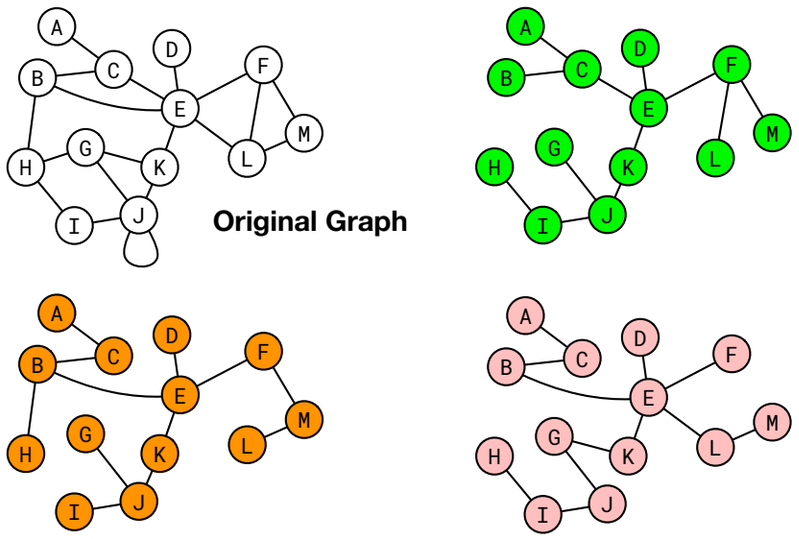
Spanning trees are useful in a few ways. There are path finding problems, like <next build>

“**can** I get from this node to that node”,

or <next build>

“what is the **shortest path** from this node to that node”. And spanning trees can answer those kinds of questions.

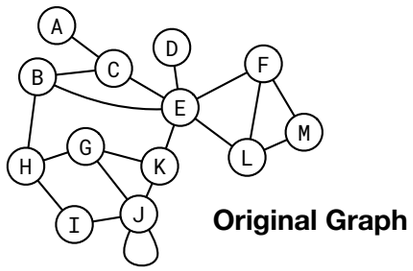
Many Possible Trees



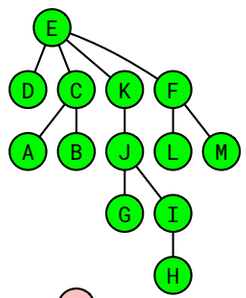
For any graph with more than two connected nodes, there will be several possible trees. Lots of them, it turns out. Given this original graph, <next build> here are three possible spanning trees for this graph. Just take a second to look at them. It is the same set of nodes, but only some of the edges. They're all completely valid spanning trees.

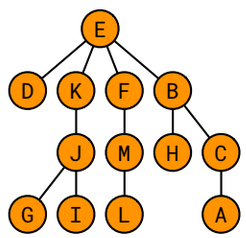
And to reinforce that they really are trees, I've moved them around so you can see them as trees, all starting from the node E.

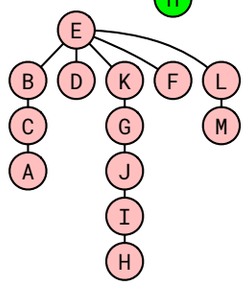
Many Possible Trees



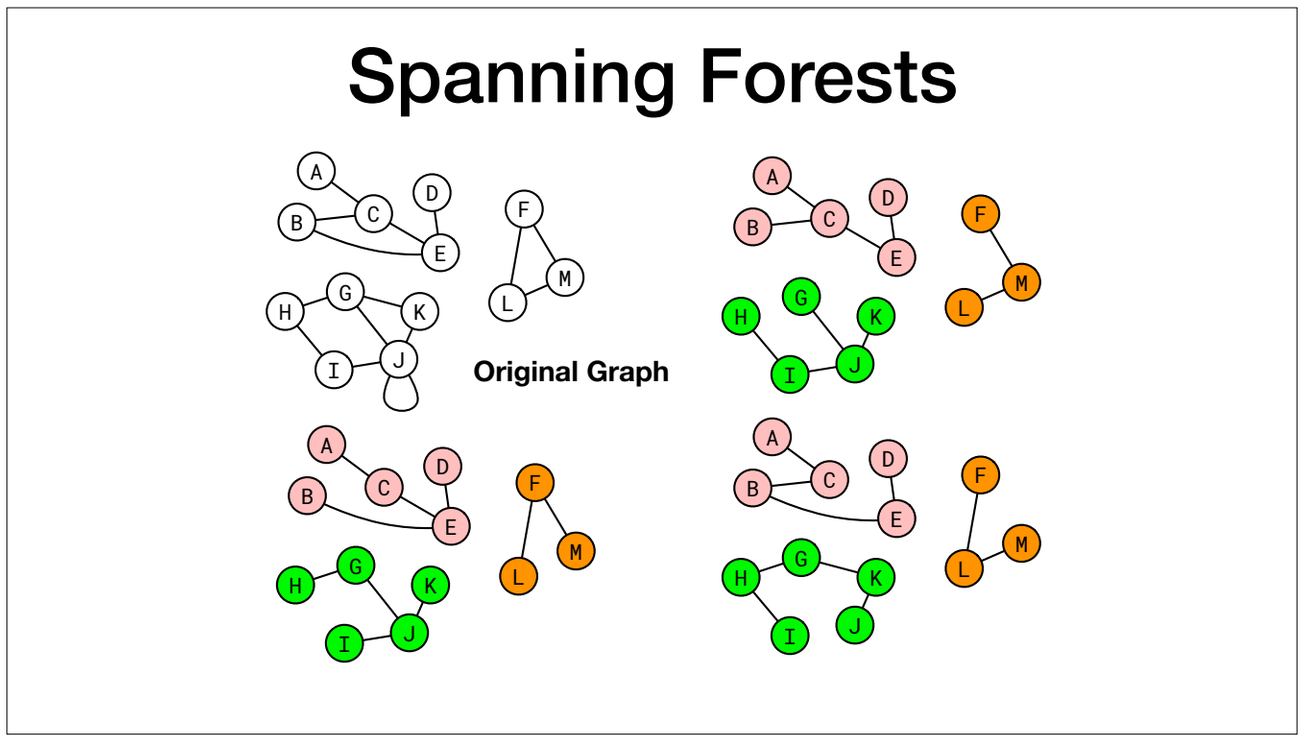
Original Graph







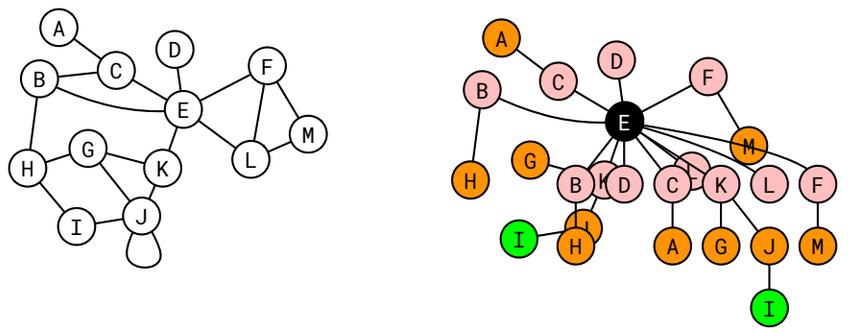
E is always at the top, but depending on which spanning tree you look at, a node like A or B could be at a different rank.



Now, if our graph has some islands of nodes that aren't connected to others, it isn't possible to form a single spanning tree, since a spanning tree technically has to involve all nodes in the graph. **<next build>**

For that situation, you can make a _spanning forest_, which treats each island component of the graph as its own mini-graph. This just shows three different possible forests for the original graph.

Minimum Spanning Tree (MST)



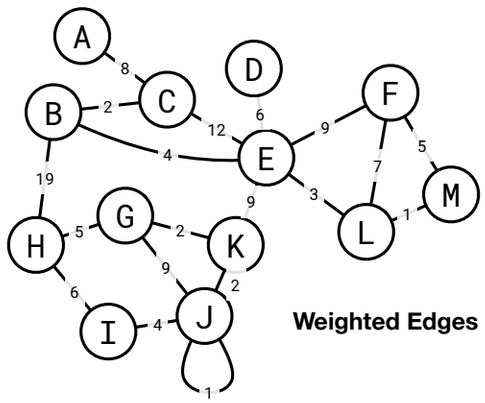
There's a special kind of spanning tree, called the minimum spanning tree, which can help optimize lots of different kinds of activities, like network packet routing, robot movement, and lots of other business use cases.

Here we’re showing the minimum spanning tree rooted at E.

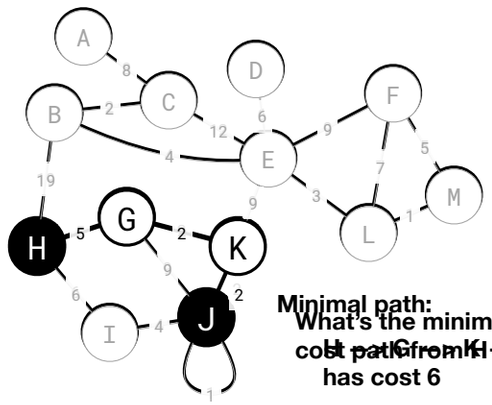
<next build> and this is the tree re-drawn to make the height a bit more obvious.

The minimum spanning tree is just one of the spanning trees that has the smallest possible tree height. In this case there are at most three edges from the root to lowest leaf. The minimum spanning tree might be unique, but if not, it's a tie.

MST With Weights



Weighted Edges



Minimal path:
H, G, K, J

What's the minimum cost path from H to J?

~~cost path from H to J?~~

has cost 6

Check out Dijkstra's Algorithm for more on path finding on weighted graphs.

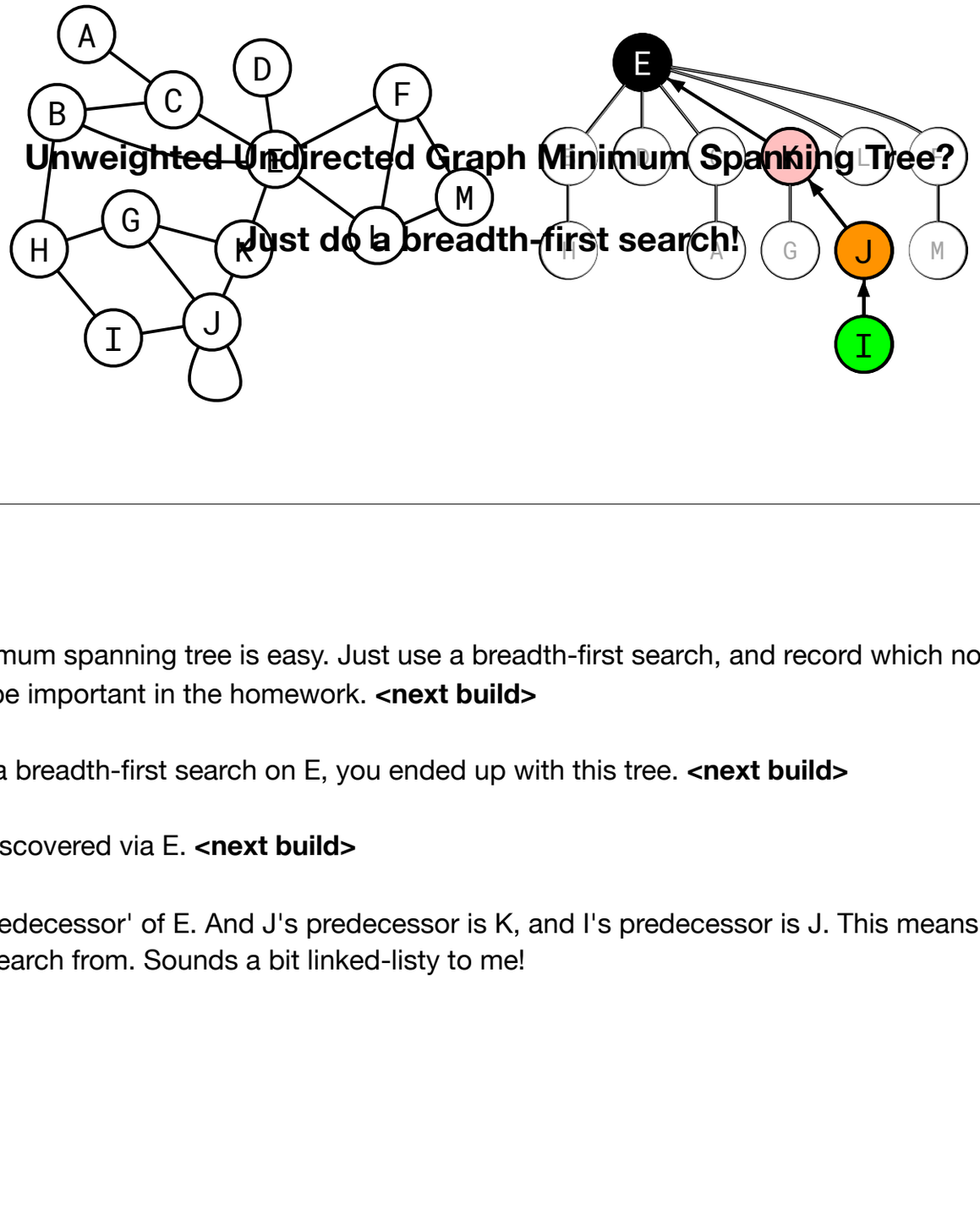
One bit of metadata you can add to your edges is a weight value. So if this graph represents towns, the edge weight could represent how much time it takes to make the trip between towns. I dunno, maybe there are mountains or something in between B and H.

If you have a weighted graph like this, finding the minimum spanning tree or shortest path becomes a little harder. We're not going to delve into that in this class, so just be aware that weighted graphs require special treatment.

<next build> To give yourself just a taste of what's involved, what do you think the minimum cost path is from H to J? And by that, what's the path from H to J that minimizes the sum of each edge you take? … pause… **<next build>**

If you went through node K, you're right. When you have weights, it's not just a simple matter of looking at the number of edges. **<next build>** Check out Dijkstra's algorithm if you're interested to see more about this.

Node Ancestry



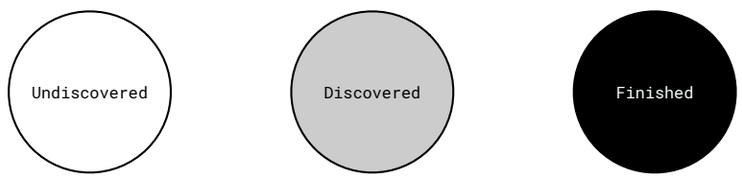
For an an unweighted graph, finding the minimum spanning tree is easy. Just use a breadth-first search, and record which node each node was discovered from, and the resulting tree will be a minimal one. This will be important in the homework. **<next build>**

So if you have this graph again, and you run a breadth-first search on E, you ended up with this tree. **<next build>**

All of the pink nodes, B, D and so on, were discovered via E. **<next build>**

So node K for example have a 'parent', or 'predecessor' of E. And J's predecessor is K, and I's predecessor is J. This means from any node you can follow the ancestry all the way back to the node we started the search from. Sounds a bit linked-listy to me!

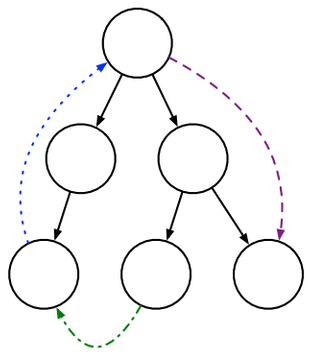
Classifying Edges



When you perform a depth-first search, you can also classify edges based on the discovery state of the nodes on each end of the edge. This is also something you'll do in the homework.

Initially, all nodes are considered undiscovered, we say they're all white. When you discover a node, you color it gray, and when you are completely done exploring a node you color it black. You also can mark the discovery and completion time, using an incrementing ‘clock’ value.

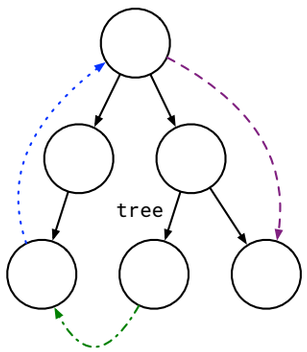
Classifying Edges



Spanning tree edge classification during depth-first search

You use the discovery information to to classify edges into four types: tree, back, forward and cross edges, indicated with different graphical styles here.

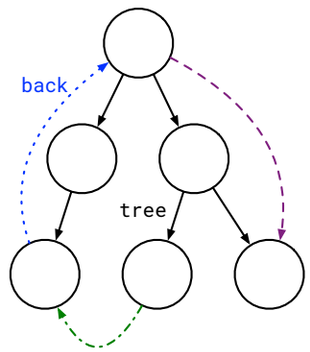
Classifying Edges

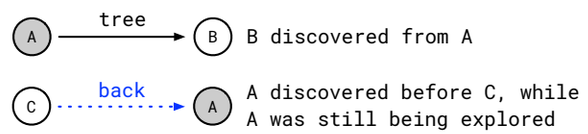


Spanning tree edge classification during depth-first search

- A tree edge from A to B indicates that B was discovered via A.

Classifying Edges

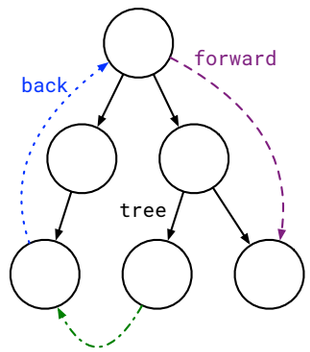


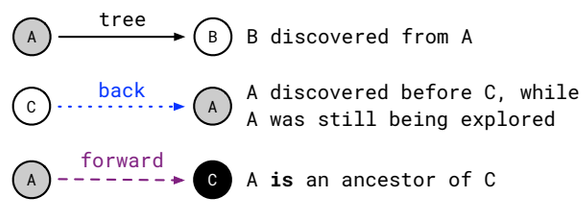


Spanning tree edge classification during depth-first search

- A back edge from C to A indicates that A was discovered before C, and C was discovered while A was still being explored. If we don't get any back edges, then we know the graph is acyclic!

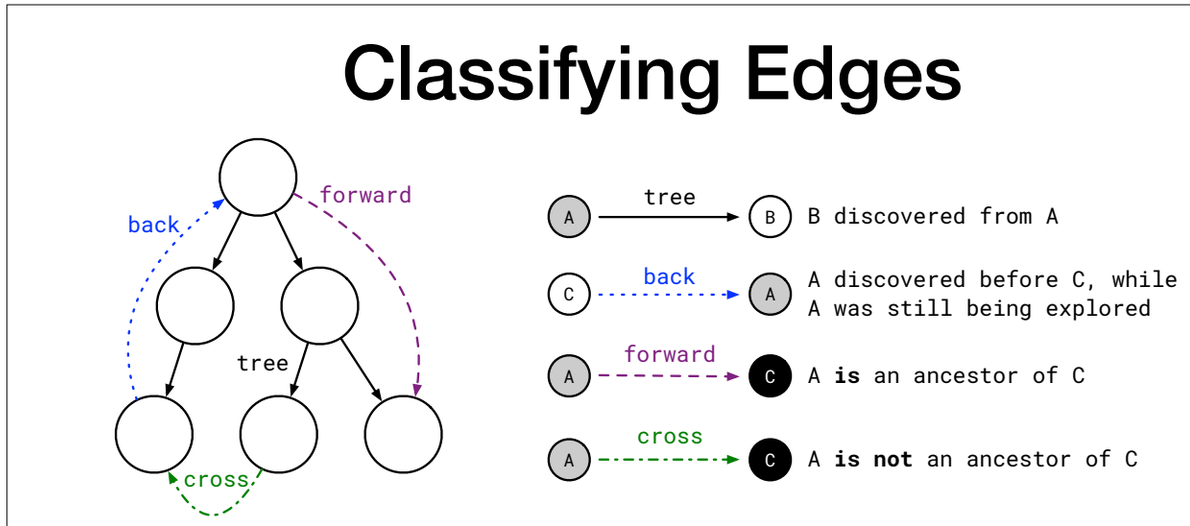
Classifying Edges





Spanning tree edge classification during depth-first search

- A forward edge from A to C indicates that C was completely examined when we found it, and A is an ancestor of C in the DFS spanning tree.

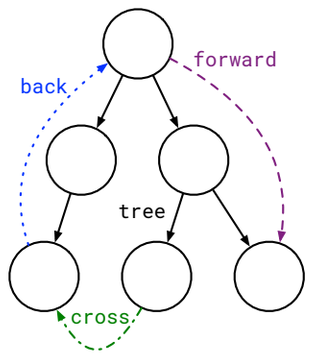


- A cross edge from A to C indicates the C was completely examined when we found it, and A is NOT an ancestor of C in the DFS spanning tree.

To determine edge types, you need to use metadata about of the start and end nodes that is set during the search process. You can pull this off using color only, or using discovery and completion time.

Edge Recap

Use discovery color or time to determine edge type.



- **Tree:** end node is white
- **Back:** end node is gray
- **Forward:** end node is black, start node is ancestor of end node
- **Cross:** end node is black, start node is NOT an ancestor of end node

So to recap: use the discovery information, either color or time, to determine the edge type.

Tree edges are when the end node is white, back edges are when the end node is gray.

For forward and cross edges, the end node is already black when we traverse the edge. In that case, you need to look at ancestry information. If the start node is an ancestor of the end node, it's a forward edge. If the start node isn't an ancestor of the end node, it's a cross edge.

And with that, you should have enough to get going on the homework!

Episode 9

Graph Homework

This episode is a little different from what I've recorded so far. All the slides in this episode are meant to stand alone, without me talking away on top. The intention here is that you get the slides and step through them to get super familiar with the classes associated with the homework, and what the methods all do.

Three classes to implement - see graph.hpp

Graph

Holds Node and Edge objects that define the data and structure. Gives access to the DFS and BFS operations.

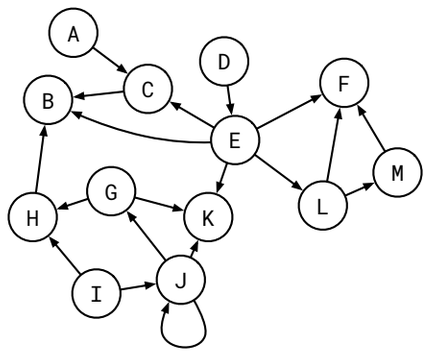
Node

Holds data (a string). DFS sets discovery and finish times. BFS sets discovery rank.

Edge

A relation between two Nodes. Doing a DFS sets the edge *type*.

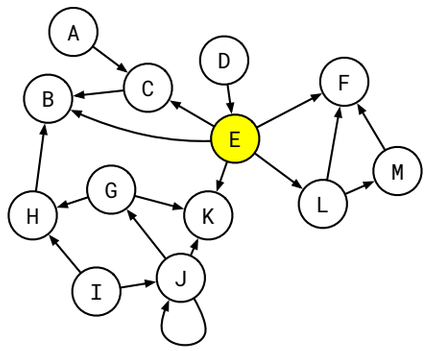
Edit graph.cpp to implement missing methods.



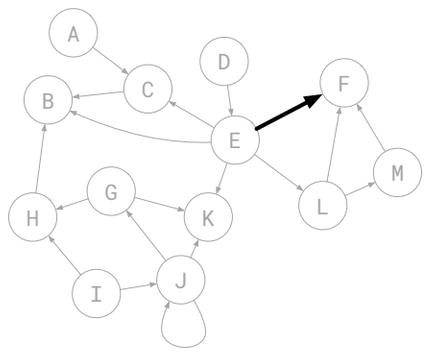
Whole thing is a graph.

No ‘start’ or ‘first’ node, unless our application semantics dictate that.

Generally we could choose to begin graph exploration at any of these locations.

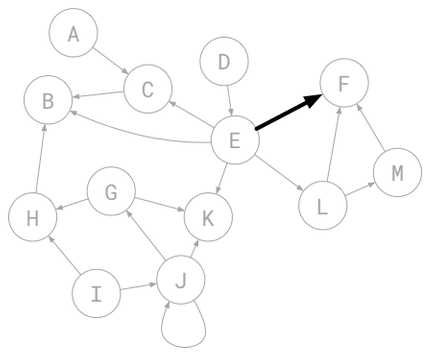


A single node contains data (a string in the case of the homework). DFS and BFS operations assign metadata including discovery time, rank, color, and which node was its predecessor in the case of a depth-first search.



An Edge is indicated with an arrow. It holds references to the Node objects it joins.

Edges can be directed (distinct start and end, indicated with arrows), or undirected (no distinct start/end, no arrows).



Importantly, our Edges don’t know if they are directed or not. That information is actually a property of the graph, in its ‘directed’ member variable.

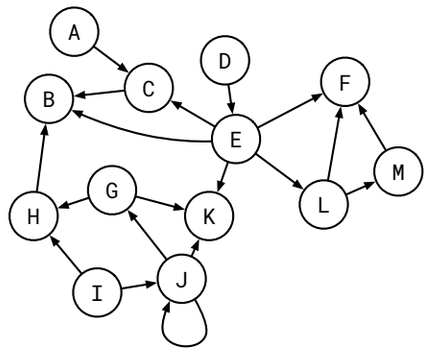
During a depth-first search we’ll add metadata to edges that describe the role they play in the search.

graph.hpp

the guided tour

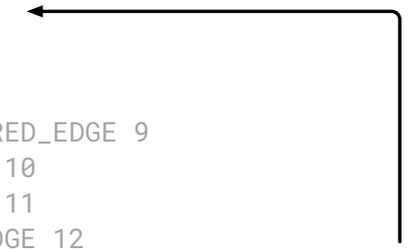
This is a detailed rundown of the contents of the header file. It includes some constants, and three classes: Graph, Node, and Edge.

Graph



First up: some constants for colors and edge types, and then a rundown of the Graph class definition.

```
#define WHITE 1
#define GRAY 2
#define BLACK 3

#define UNDISCOVERED_EDGE 9
#define TREE_EDGE 10
#define BACK_EDGE 11
#define FORWARD_EDGE 12
#define CROSS_EDGE 13
```



Nodes are *colored* according to their discovery state.

White = Not (yet) discovered
Gray = Discovered but not (yet) finished
Black = Finished

done for you

```
#define WHITE 1
#define GRAY 2
#define BLACK 3

#define UNDISCOVERED_EDGE 9
#define TREE_EDGE 10
#define BACK_EDGE 11
#define FORWARD_EDGE 12
#define CROSS_EDGE 13
```

Edges are classified by type according to the graph state when they were discovered during a DFS.

done for you

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;

public:
  // will show next
};
```

‘directed’ tells us if the Edges have distinct start/end nodes or if they make no such distinction.

done for you

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;

public:
  // will show next
};
```

‘nodes’ and ‘edges’ contain references to Node and Edge objects. These model the graph’s data and structure of that data.

done for you

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;

public:
  // will show next
};
```

‘search_edges’ & ‘search_nodes’ contain information about an ongoing or the most recently completed BFS or DFS execution. The one for edges is not necessary for this assignment, but some algorithms can use it.

The one for nodes is used as your queue or stack for BFS and DFS respectively.

done for you

```
class Graph {
private:
  bool directed;

  vector<Node*> nodes;
  vector<Edge*> edges;

  vector<Edge*> search_edges;
  vector<Node*> search_nodes;

  int clock;

public:
  // will show next
};
```

‘clock’ should be used to keep track of the current time step in a depth-first search.

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

Constructor and Destructor

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

get node and edge vectors.

Used to query state of graph after a search. Node and Edge discovery information should be kept around after a search, so traversing these vectors gives us the results of the search.

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

Housekeeping functions to query the clock, and to add/ remove edges and nodes.

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

Accessor and mutator methods that query and set the edges to directed or undirected.

Again: this is a property of the graph, not of individual edges, though other implementations might do it differently.

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

Supplies you with a set of edges that are related to this node. For undirected graphs, this includes any edge that touches the node; for directed graphs it only includes edges *leaving* the node.

done for you

```
class Graph {
private:
  …
public:
  Graph();
  ~Graph();
  vector<Node*> getNodes();
  vector<Edge*> getEdges();
  int getClock();
  void addNode(Node& n);
  void addEdge(Edge& e);
  void removeNode(Node& n);
  void removeEdge(Edge& e);
  bool isDirected();
  void setDirected(bool val);
  set<Edge*> getAdjacentEdges(Node& n);
  friend std::ostream &operator
      << (std::ostream& out, Graph graph);
};
```

Convenience method that lets you toss a Graph object into an output stream and get something useful to happen.

cout << my_graph << end;

Notice this is a ‘friend’ function, so it is not defined in the Graph namespace. We’re just saying to the compiler that it is cool with us if this function uses the Graph’s private member variables.

done for you

```
class Graph {
private:
  …
public:
  …
  void clear();
  void tick(string message);

  void dfs(Node& start);


  void bfs(Node& start);
  void bfs(Node& start, Node& target);
};
```

Resets all nodes to WHITE, with -1 discovery and finish times and rank. Resets all edges to UNDISCOVERED_EDGE. Resets clock to 0.

Testing code will call this as needed. Don't call this yourself.

implement this

```
class Graph {
private:
  …
public:
  …
  void clear();
  void tick(string message);

  void dfs(Node& start);


  void bfs(Node& start);
  void bfs(Node& start, Node& target);
};
```

OPTIONAL debugging method.
Read the header documentation for more info.

implement this

```
class Graph {
private:
  …
public:
  …
  void clear();
  void tick(string message);

  void dfs(Node& start);


  void bfs(Node& start);
  void bfs(Node& start, Node& target);
};
```

Execute a depth-first search from the indicated start node. It searches all reachable nodes.

At the end, all explored nodes are BLACK and have correct discovery/exit time, all unreachable nodes are WHITE. All edges have correct edge types (unfollowed edges should remain UNDISCOVERED.)

For DFS, mark nodes GRAY when we first discover them, and BLACK when we exit (finish) them.

implement this

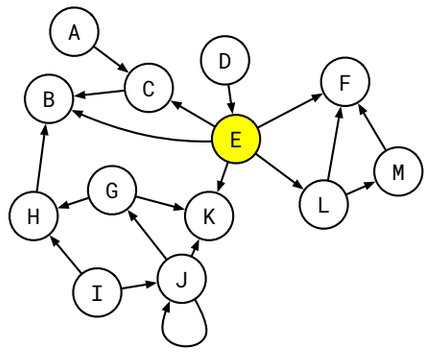
```
class Graph {
private:
  …
public:
  …
  void clear();
  void tick(string message);

  void dfs(Node& start);


  void bfs(Node& start);
  void bfs(Node& start, Node& target);
};
```

Execute a breadth-first search starting from the given node. This sets the ‘rank’ value on all nodes to something appropriate: -1 for unreachable nodes, 0 for the start node, 1 for nodes that are one edge from the start node, and so forth.

For a BFS, mark nodes GRAY when they are enqueued, and BLACK when they are dequeued.

implement this

Node



Now we'll look at the Node class definition.

```
class Node {
private:
  string data;
  int color;
  int discovery_time;
  int completion_time;
  int rank;
  Node* predecessor;

public:
  Node(string s);
  ~Node();
  string getData();
  void setData(string s);
  void setRank(int rank);
  friend std::ostream &operator
     << (std::ostream& out, Node node);
  …
};
```

This is the part of the Node class that is done for you. Don't worry about setting the 'data' variable, but you are responsible for setting (or clearing) the other variables.

done for you

```
class Node {
private:
  …
public:
  …
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
          (int& color, int& disco_time,
           int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);
};
```

Implement these functions.

‘clear’ resets the private variables to a default state.

‘setColor’ sets the node’s discovery state and (for DFS) sets the associated time value.

implement this

```
class Node {
private:
  …
public:
  …
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
         (int& color, int& disco_time,
          int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);
};
```

‘getDiscoveryInformation’ uses four pass-by-reference variables for you to set to appropriate values. While this function technically *returns* void, it is used to access four state variables at one time.

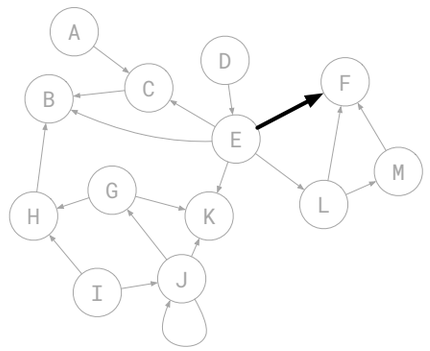
implement this

```
class Node {
private:
  …
public:
  …
  void clear();

  void setColor(int search_color, int time);

  void getDiscoveryInformation
          (int& color, int& disco_time,
           int& finish_time, int& bfs_rank);

  bool isAncestor(Node& other);
  void setPredecessor(Node& other);
};
```

These methods give information about the *spanning tree* formed during a DFS. The predecessor is the node we were on our node was discovered. An ancestor is a predecessor or any of our predecessor's predecessors.

implement this

Edge



Now we'll look at the Edge class definition.

```
class Edge {
private:
  Node* a;
  Node* b;
  int type;
public:
  Edge(Node& n1, Node& n2);
  ~Edge();
  int getType();
  Node* getStart();
  Node* getEnd();
  void setWeight(float val);
  friend std::ostream &operator
    << (std::ostream& out, Edge edge);

  void setType(int edge_type);
};
```

The Edge class is mostly done for you.

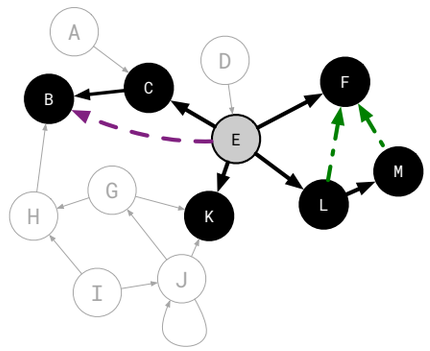
done for you

```
class Edge {
private:
  Node* a;
  Node* b;
  int type;
public:
  Edge(Node& n1, Node& n2);
  ~Edge();
  int getType();
  Node* getStart();
  Node* getEnd();
  void setWeight(float val);
  friend std::ostream &operator
    << (std::ostream& out, Edge edge);

  void setType(int edge_type);
};
```

All you need to implement the ‘setType’ method (and that should only be one line long).

implement this

Algorithms



Now we'll look at the basic pseudocode for the two most common graph algorithms.

Breadth-First Search

```
bfs(start):
  clear graph
  mark start gray
  Q = empty queue
  add start to Q
  while Q has stuff in it:
    v = popped element from Q
    mark v black
    visit v
    for each unmarked neighbor w
        mark w gray
        add w to Q
```

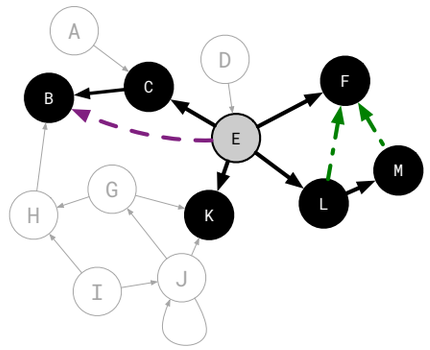
implement this

Depth-First Search

```
# clear graph before initial call
dfs(node):
  mark node gray
  visit node
  E = edges related to node
  for all edges e in E:
    a = end of e that isn't node
    if a is unmarked:
      dfs(a)
  mark node black
```

implement this

Advice



Rely on the unit tests. Make sure that the self-contained, easy methods pass the tests before moving on to more complex things like the DFS and BFS routines.

Good Luck!