Graphs and Graph Algorithms.

This is not the whole story.

There are algorithms for linked lists and binary trees that you have hopefully become familiar with: in a linked list, you use a *while loop* and a *cursor* to traverse a list.

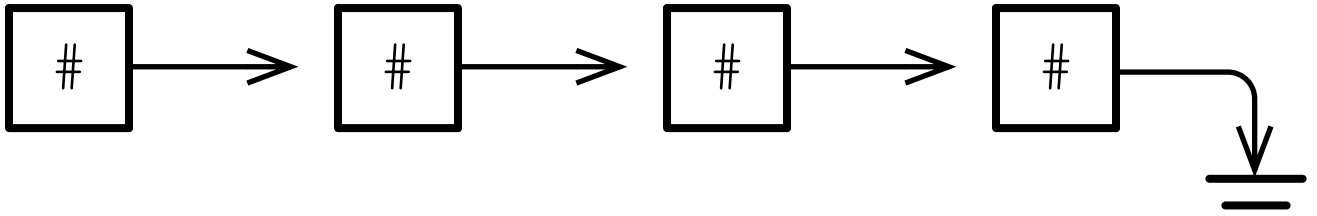
```
// linked list scan
while (cursor->next != NULL) {
  cursor = cursor->next;
}
```

With a binary tree, you can do three kinds of traversals (preorder, inorder, and postorder).

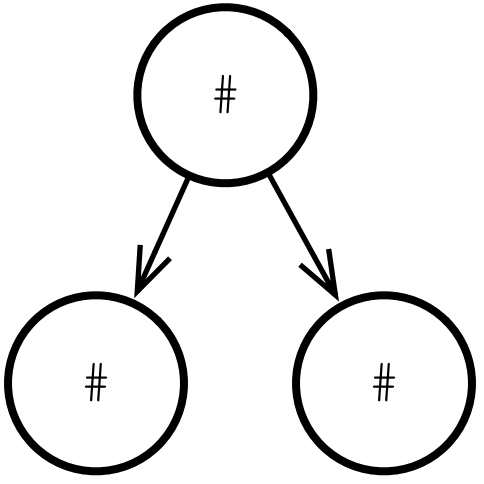
The exact details depend on what you're doing: looking for a particular value, counting nodes, looking for the end or the list of a leaf node, and so on.

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

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

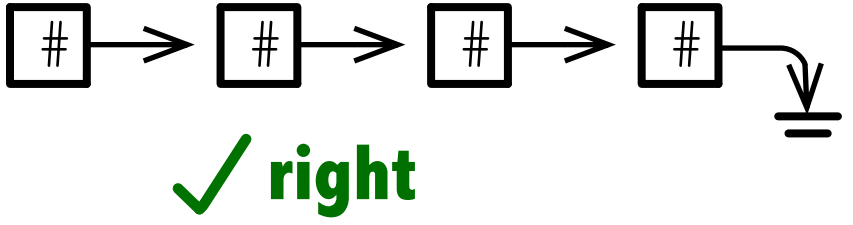


A **Binary Tree** is a bit more complicated. A binary tree's nodes contain data and *two* links: a left child and a right child.



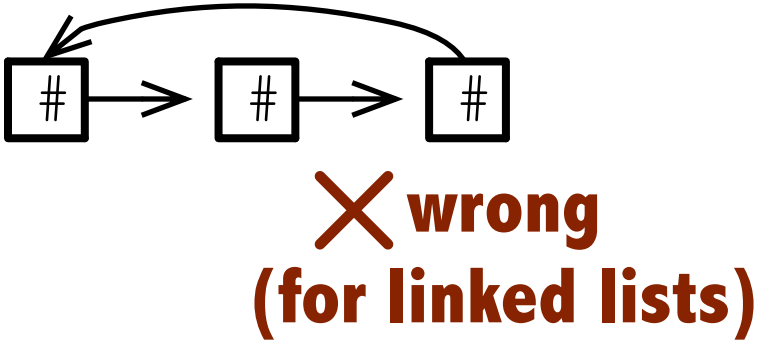
You can think of both of these as very simple graphs that have rules about how the nodes can be related.

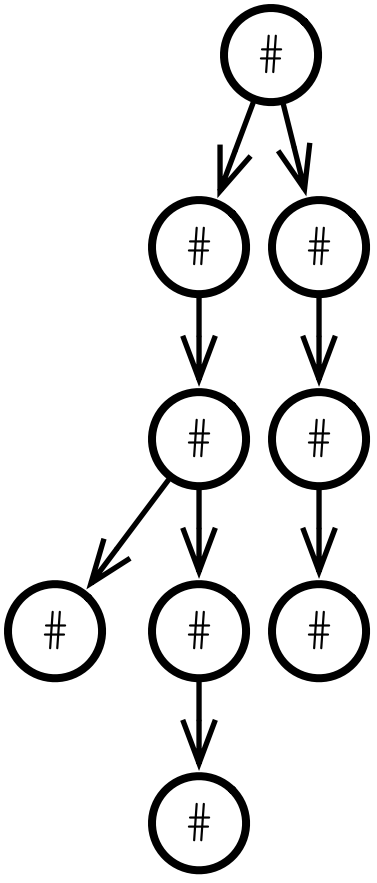
A *Graph* is a structure that contains nodes (where data lives) and edges (relationships between nodes). We can impose constraints on how we form edges, like we do with Linked Lists and Binary Trees. Here are some kinds of graphs:



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

We impose rules on the *topology:* what can be reached by following nodes. You can't backtrack like this in a linked list.

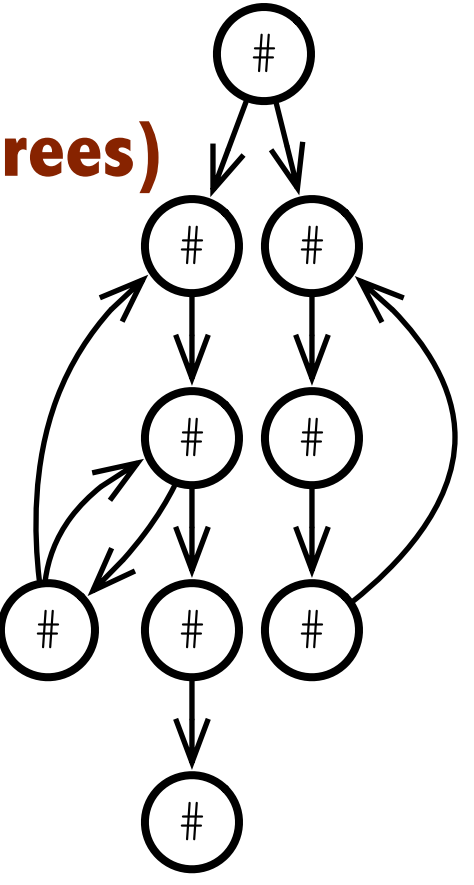




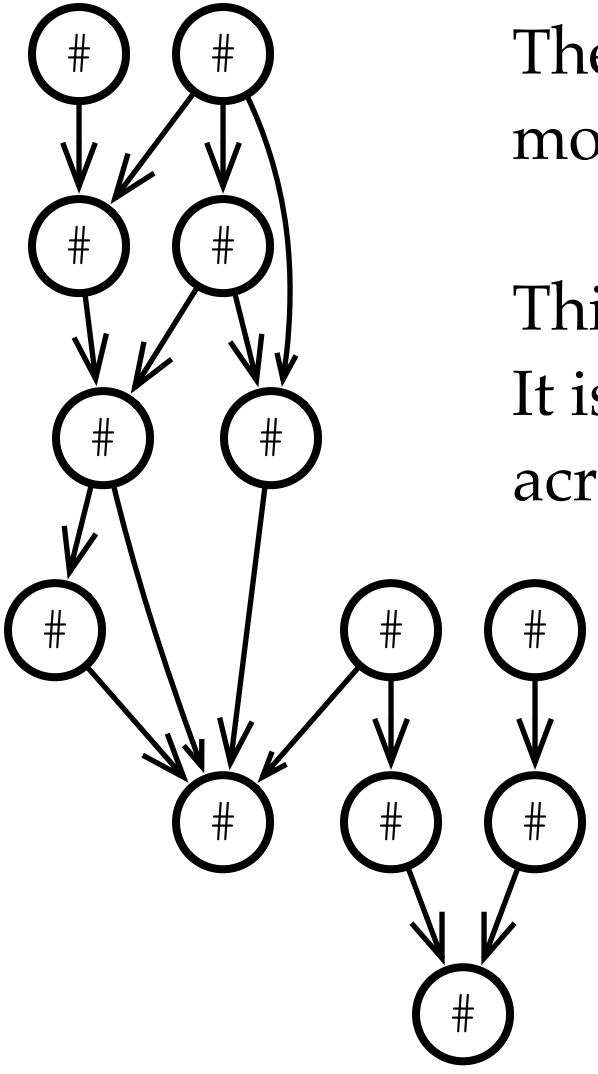
✓ right

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

✗ wrong
(for binary trees)



We've also had the implicit rule that a parent or any other ancestor can't be a child's child (don't make it weird, bro.)



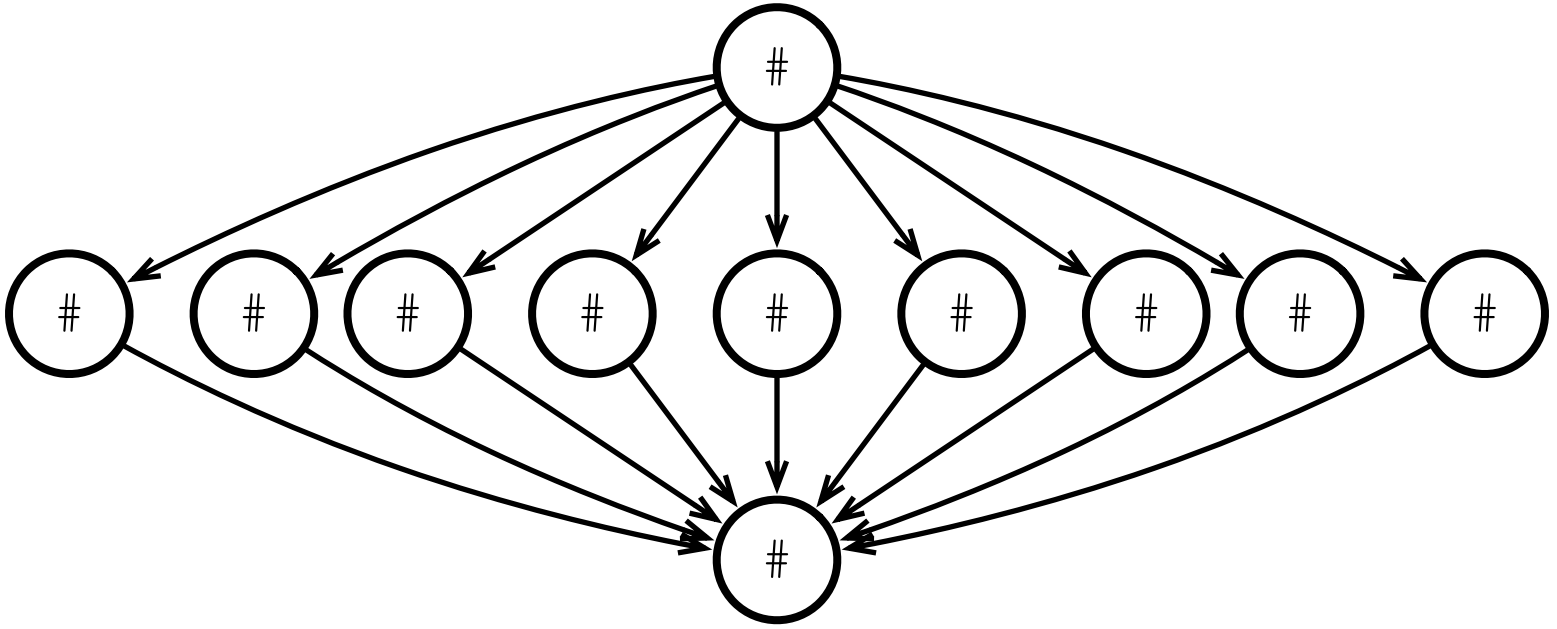
There are lots of other kinds of graphs, most of which have fewer constraints.

This one is called a ***Directed Acyclic Graph***. It is common enough that it gets its own acronym: DAG.

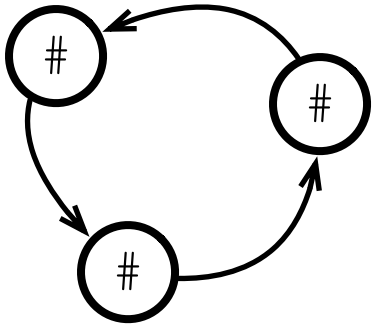
Directed: the edges have distinct start and end nodes. Indicated with arrow heads.

Acyclic: *cycles* are not allowed. 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 allow cycles, and neither do DAGs.

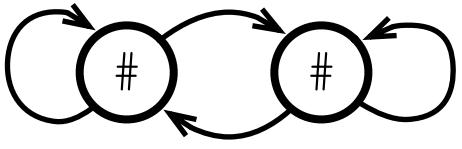
There is no rule about the number of edges that a DAG node can have leaving or entering it. In fact, there are relatively few graph types that restrict the number of edges.



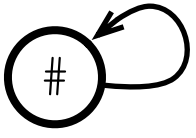
This is a perfectly legitimate (and not uncommon) DAG. From now on, assume that the number of edges is unlimited for all graph types unless otherwise noted.

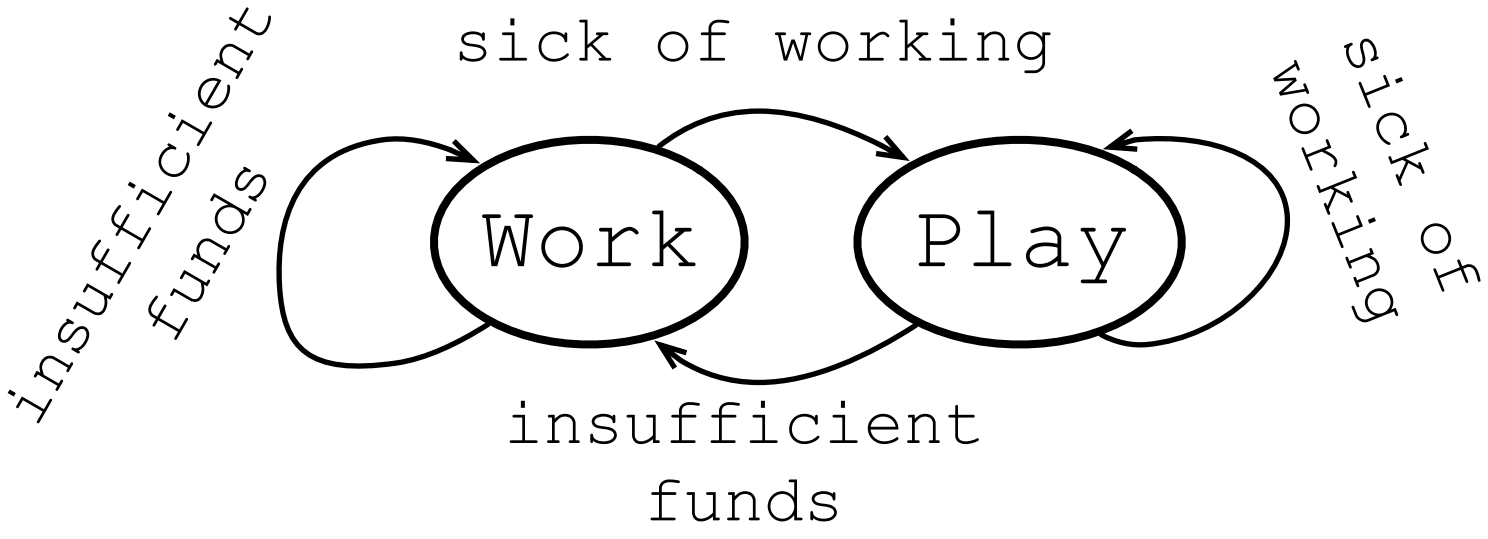


This is a **cycle**. Regardless of where you start, if you follow the edges, you will eventually get back to where you started. These aren't allowed in DAGs, but many other kinds of graphs allow them.



Many graphs exploit cycles. The graph at left has two nodes but four edges. There are two edges with different directions connecting them. You can even have crazy cycles where an edge *starts* and *ends* on a node. So in that case, you could have a one-node graph that also has a cycle. Weirdo graph!





This is a useful kind of graph called a ***Finite State Machine*** (FSM). Each node represents a *state*, and the system can only be in one state at a time. There are two possible states: *Work* and *Play*.

With a FSM, we also label the edges. These represent transitions from one state to another. There are two kinds of transitions: *Sick of Working* and *Insufficient Funds*. We only continue working as long as we don't have enough money. When we are sick of working, we play. We play as long as we are still sick of working, but will start working when we run out of money.

Finite State Machines are astoundingly useful tools when programming hardware. Say we have a robot that uses a camera to get eye contact with you. When it gets eye contact, it moves towards you. If you still have eye contact when it is near, it will do a dance. If you break eye contact, it runs away. In either case, it will resume looking for a friend to make eye contact with afterwards.

We have some states:

- **looking** for eye contact
- **moving** towards friend
- **near** friend
- **dancing**
- **running** away

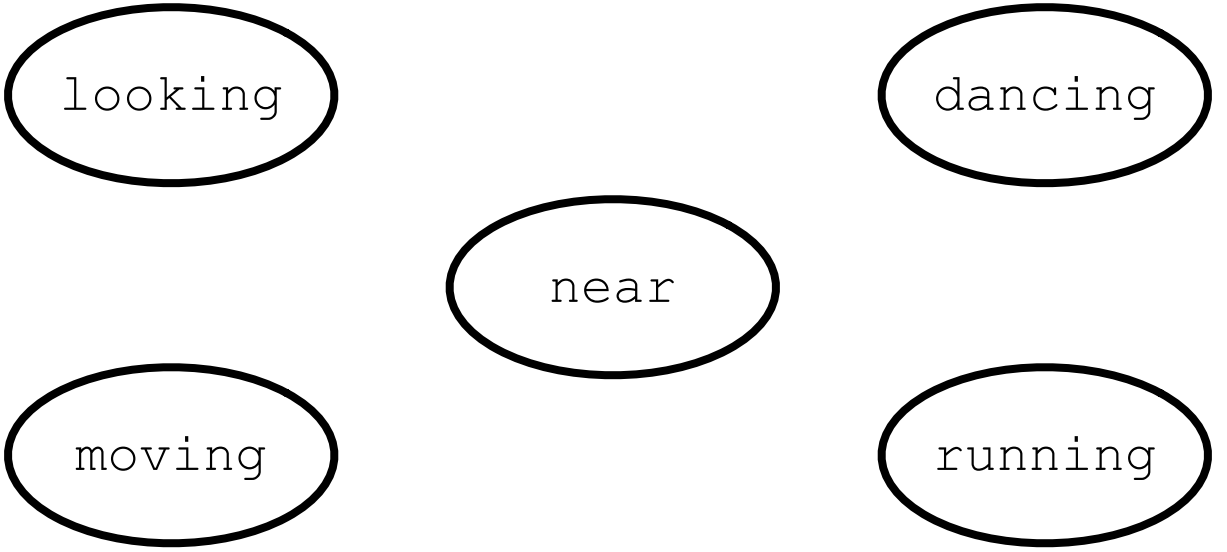
We also have events:

- **no friend** found
- **found** an eye contact friend
- **move** towards friend
- **arrive** at friend
- eye contact **broken**

Turn States/Events into an FSM

- **looking** for eye contact
- **moving** towards friend
- **near** friend
- **dancing**
- **running** away

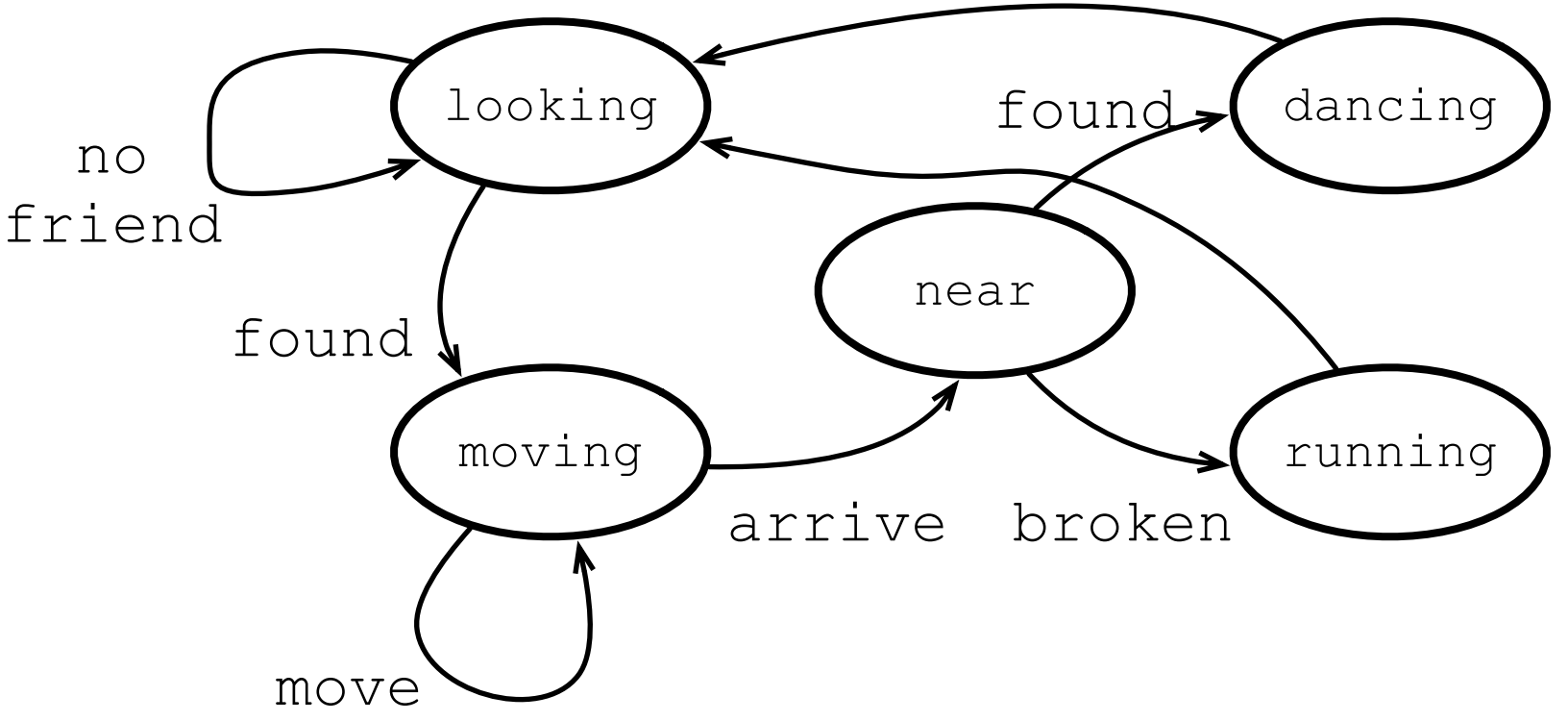
- **no friend** found
- **found** an eye contact friend
- **move** towards friend
- **arrive** at friend
- eye contact **broken**



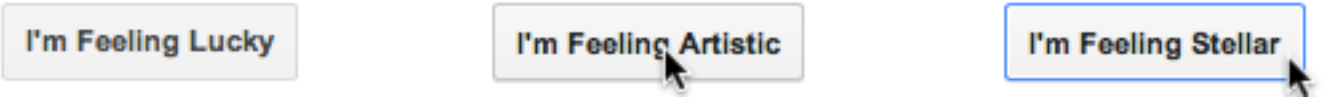
Turn States/Events into an FSM

- **looking** for eye contact
- **moving** towards friend
- **near** friend
- **dancing**
- **running** away

- **no friend** found
- **found** an eye contact friend
- **move** towards friend
- **arrive** at friend
- eye contact **broken**



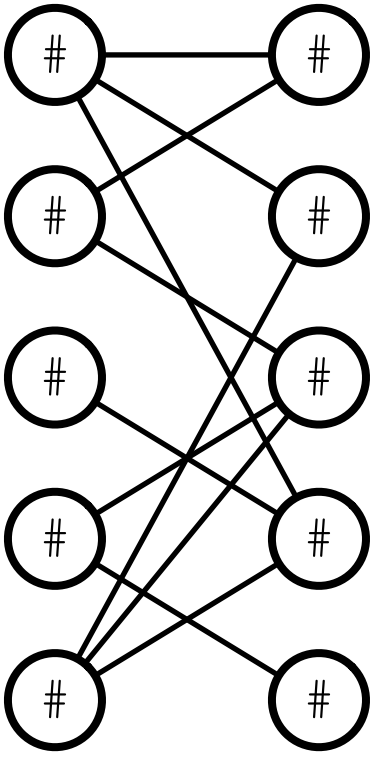
Finite State Machines are also used in programming Graphical User Interfaces (GUIs). For example, an on-screen button may have several states: bored, mouse_over, pressed, selected. The events that effect state change are mouse events: entered, exited, moved.



`bored` `mouse_over` `pressed`

A button can be one of only a few discrete states. This Google Search button changes the appearance and button text depending on mouse activity.

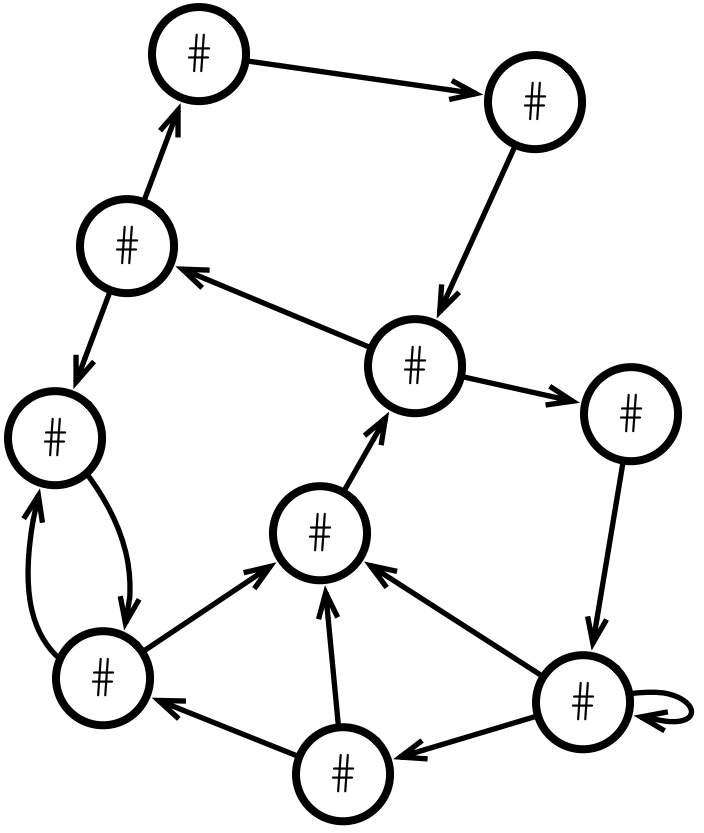
OpenGL (a graphics programming API commonly used in video games and CAD software) uses FSM semantics all over the place: it basically treats the graphics hardware as a really big finite state machine, with code providing state change events.



Here's a slightly more esoteric (to me) graph type called a ***Bipartite Graph***. These are used when you have two kinds of things (like football players and teams) and you want to relate them. Items from one group only connect to the other. We could connect players to all the clubs they played for in their careers.

This is a graph type that has constraints on which nodes can connect to which other based on set membership. There are useful algorithms for working with bipartite graphs, but we won't get into it here; I just wanted to give an example of graphs that restricted which nodes connect to which.

No 'first node' necessary.



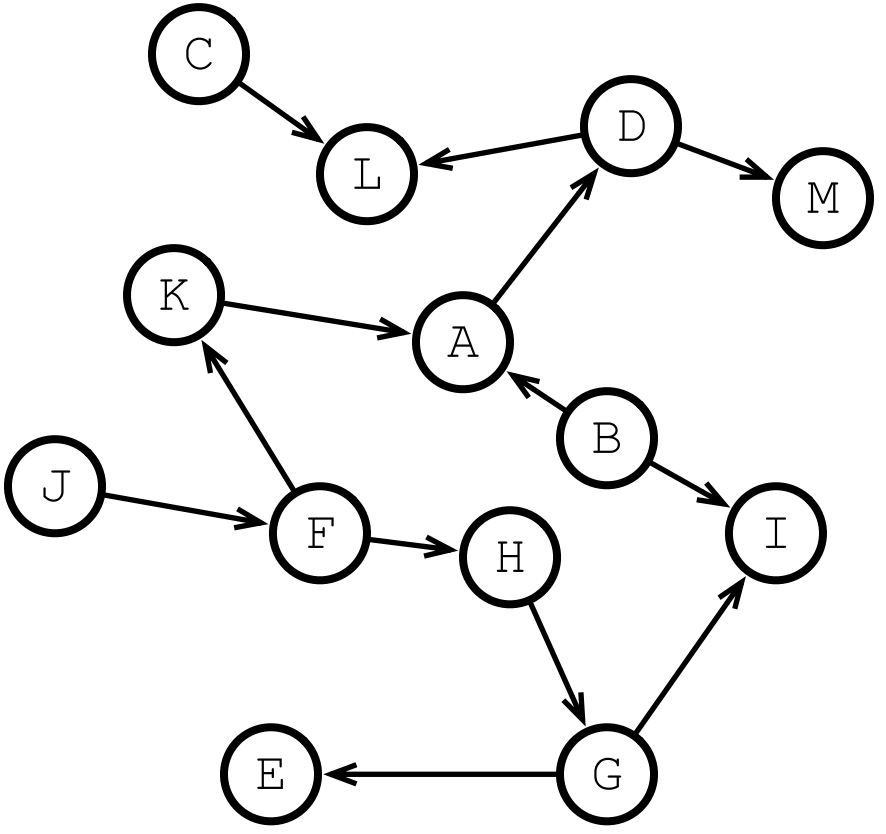
A graph does not necessarily have a 'first node'. We might have reason to choose one over another as a first node based on our use case, or we might look at the structure of the node and determine that one node sticks out as being special in some way. A binary tree has one special node: the one at the top that has no parent. But what about this one?

If there is no particular place to start, many algorithms just let you pick one arbitrarily.

Directed Graphs

A graph is directed if the edges have distinct start and end locations. This is (as far as I know) always represented with arrowheads.

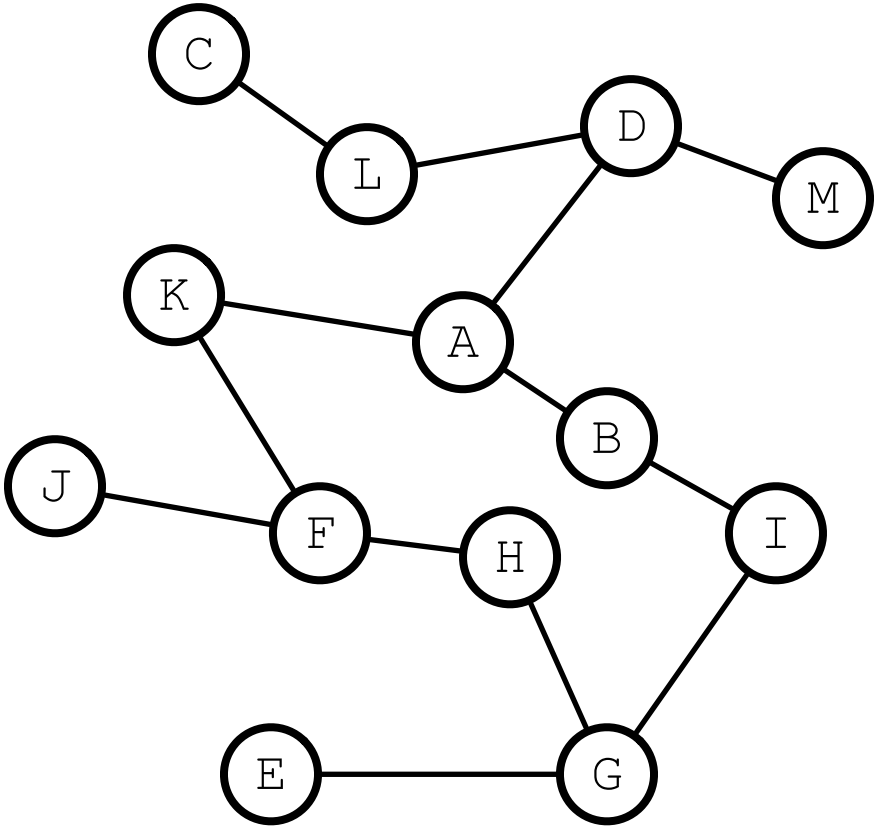
Nodes in a directed graph have a pecking order. We can move from node D to M, but not the other way around. You have to take courses in a certain order: CS 1300 precedes CS 2270, for example.



Undirected Graphs

A graph is *undirected* if the edges do not make a distinction between start and end locations. This is (as far as I know) always represented with unadorned lines.

This may represent a simple relationship where the two nodes do not have a precedence, like friendship between two people.



Graph Algorithms

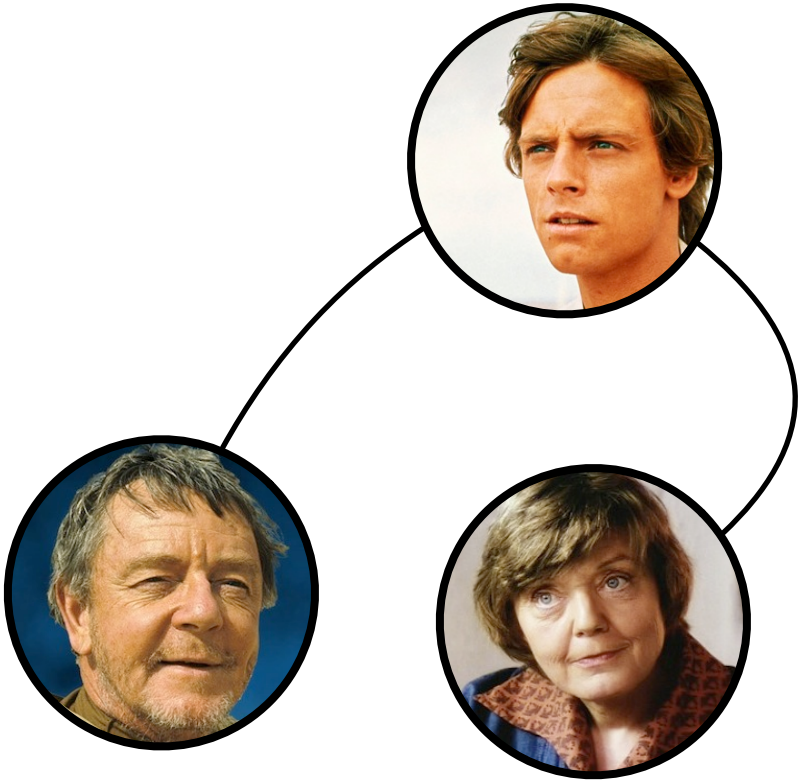
The whole point of assembling these graphs is that it lets us do computation. Can I get from *A* to *B*? How many steps away are they? What's the shortest path? The longest path?

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*.

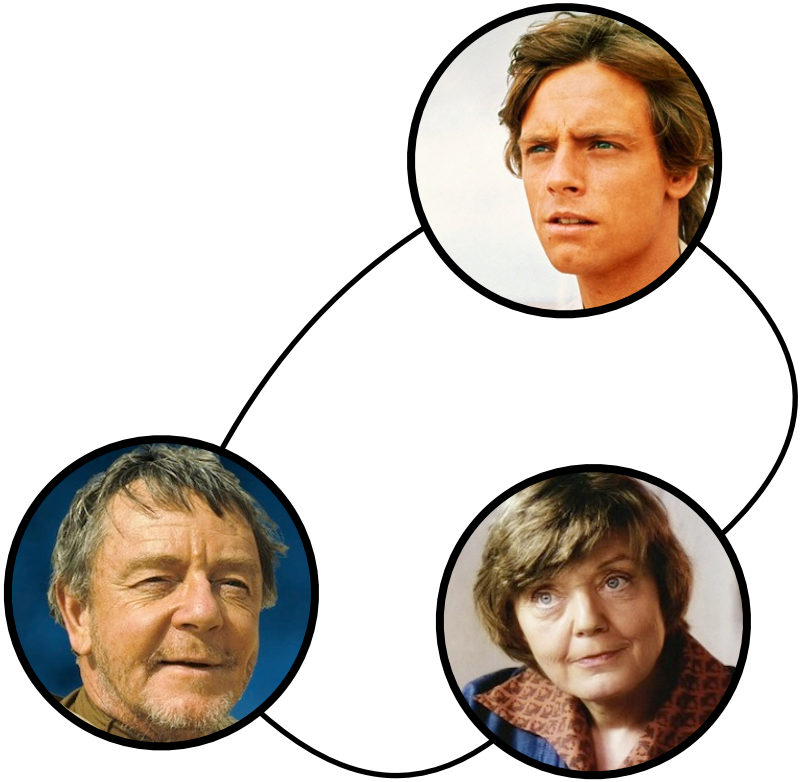
This is Luke. He's in a social network of one. Poor guy.

He lives on Tatooine with his
Uncle Owen and Aunt Beru.

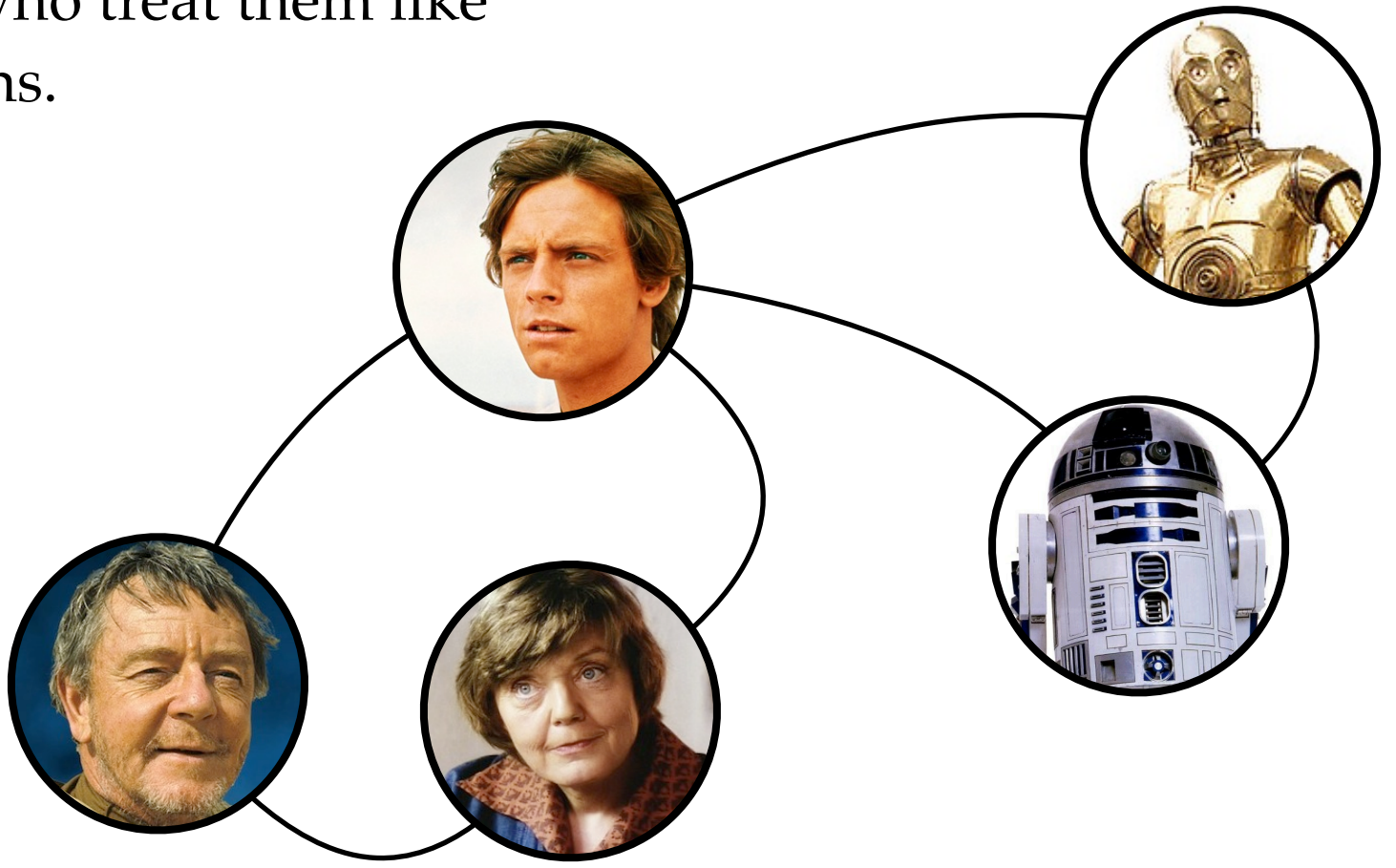


Notice that the edges are undirected: no arrowheads. In this graph, personal relationships are not directional.

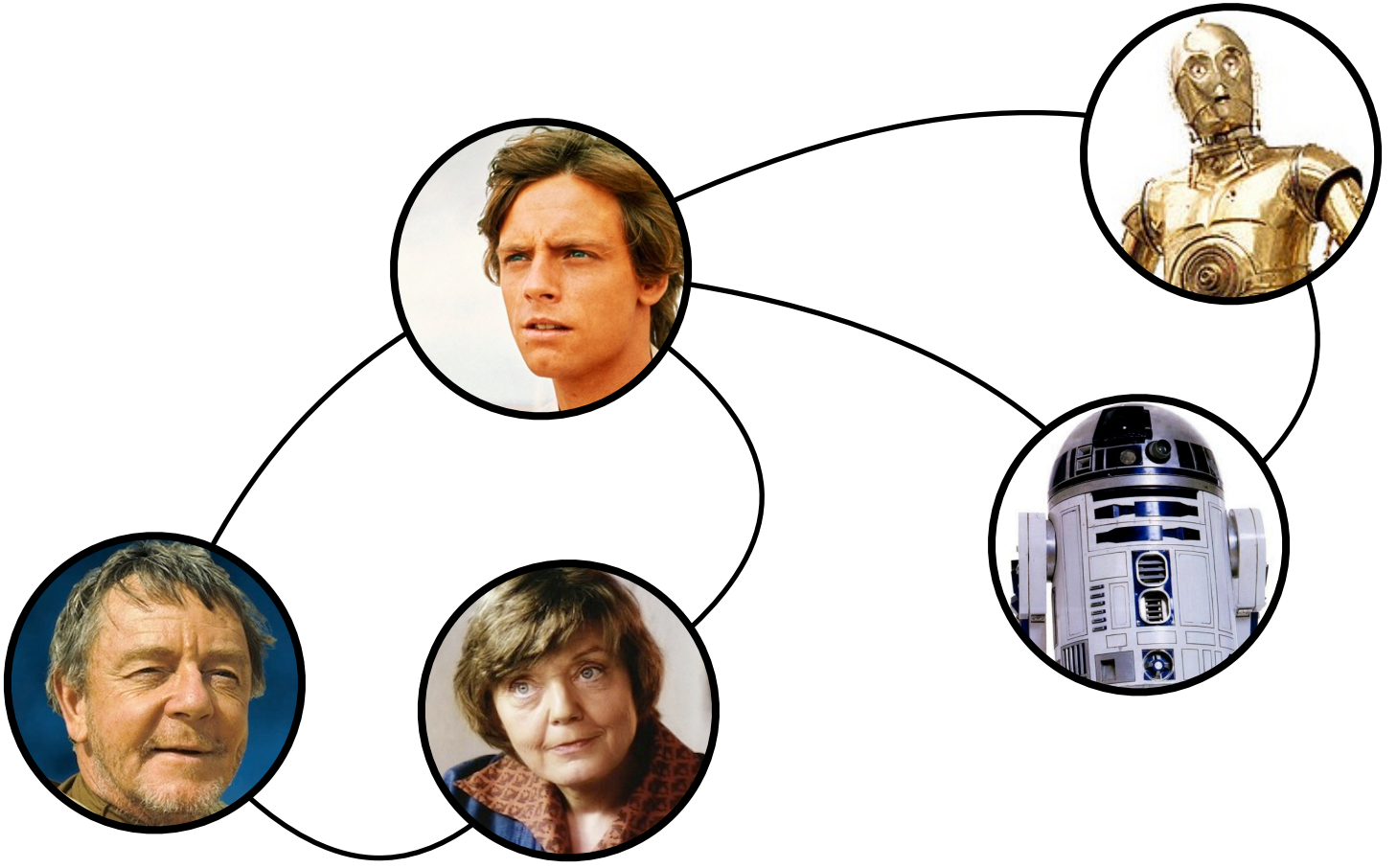
Uncle Owen and Aunt Beru are married, so we should also have an edge that connects them.

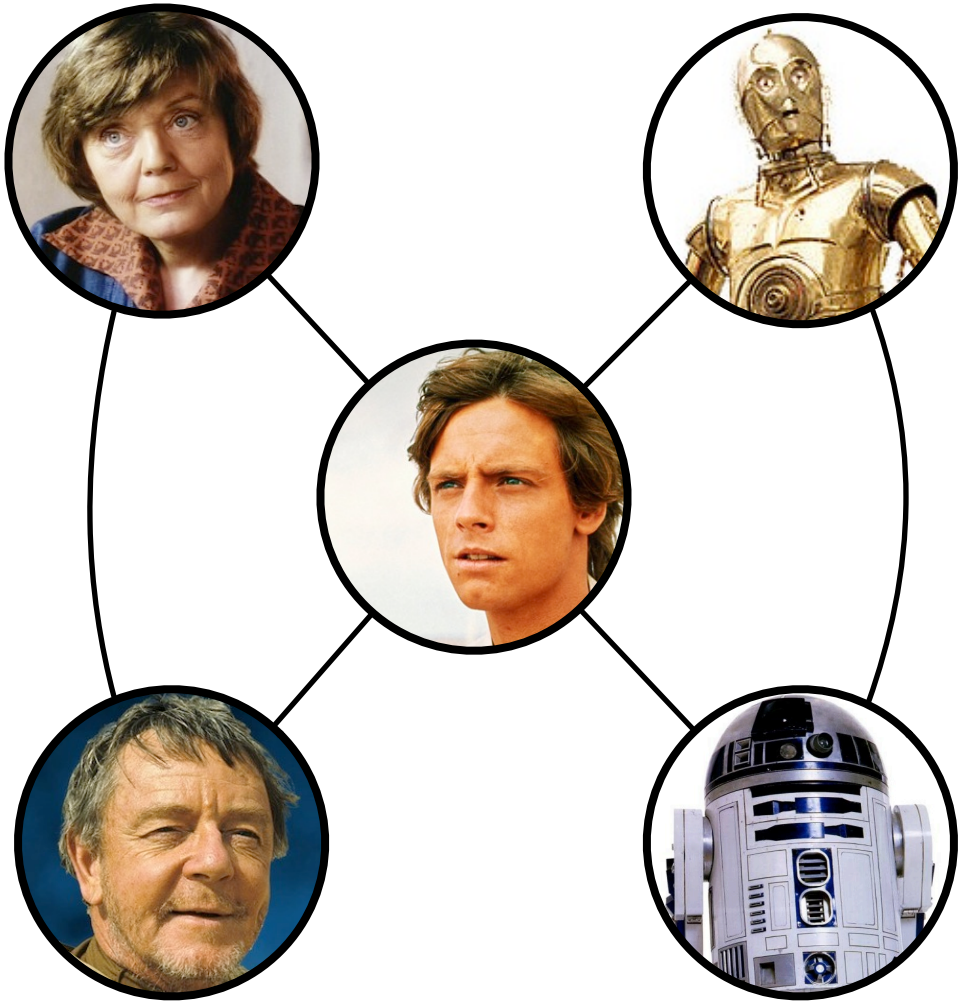


C3PO and R2D2 are buddies, and they are both (sort of) friends with Luke. But not with Owen and Beru, who treat them like toaster ovens.



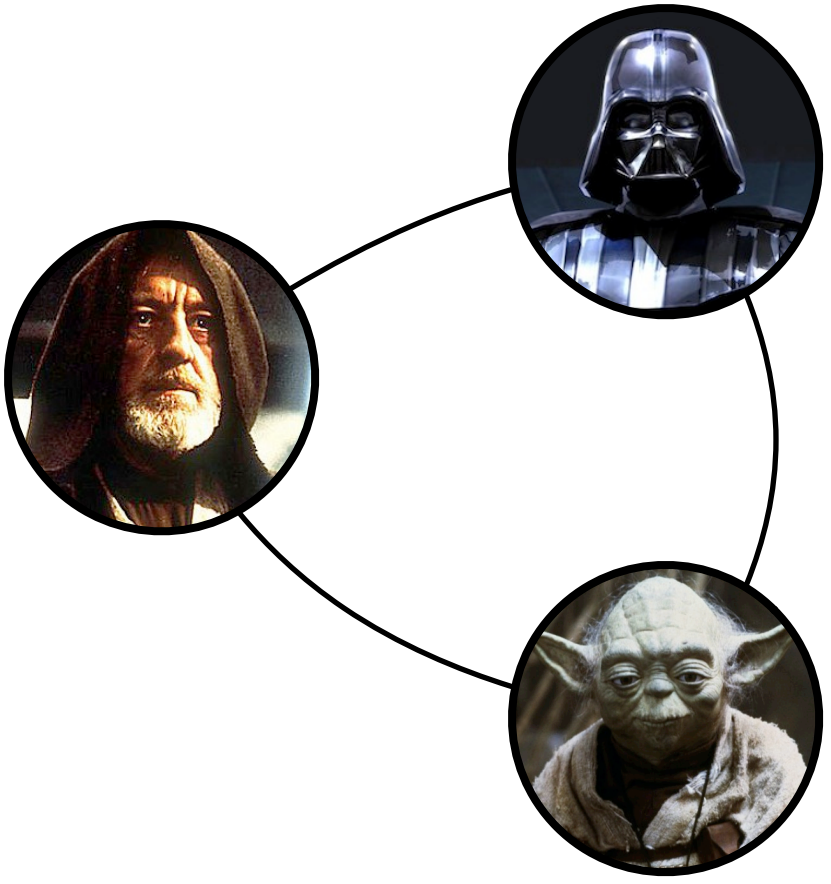
Using our eyeballs, we can see that Luke is forming the center of this social network. Everyone is connected to Luke, but the path from either droid to Uncle Owen is longer.



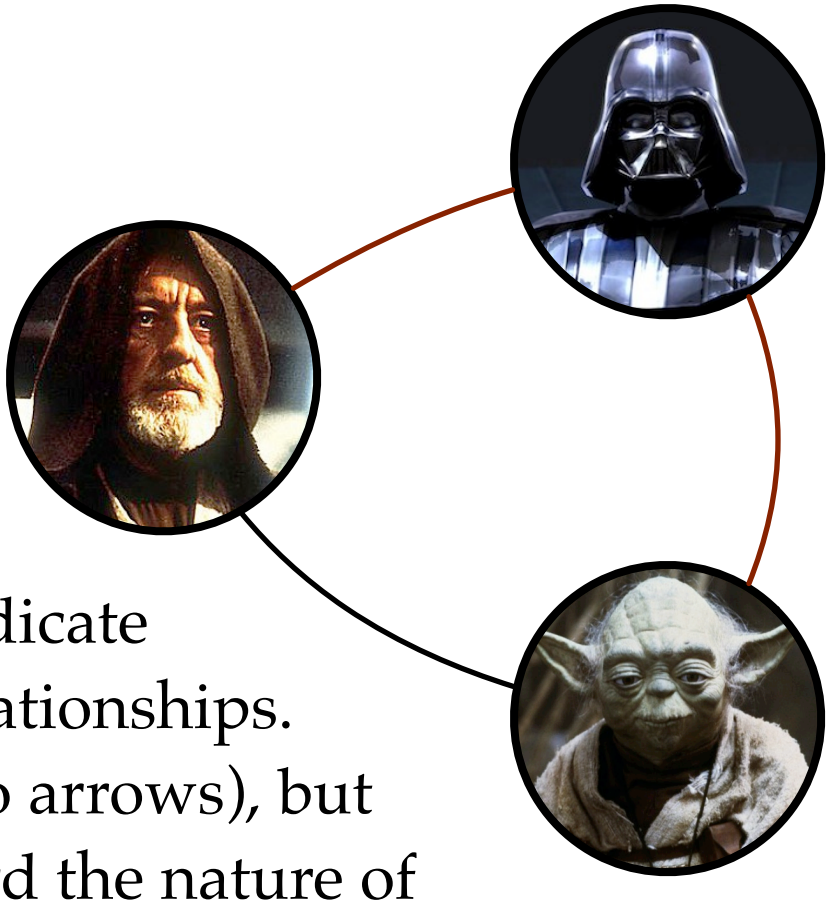


This is the same network. It doesn't matter where the nodes are. The only thing that matters is how they are connected.

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



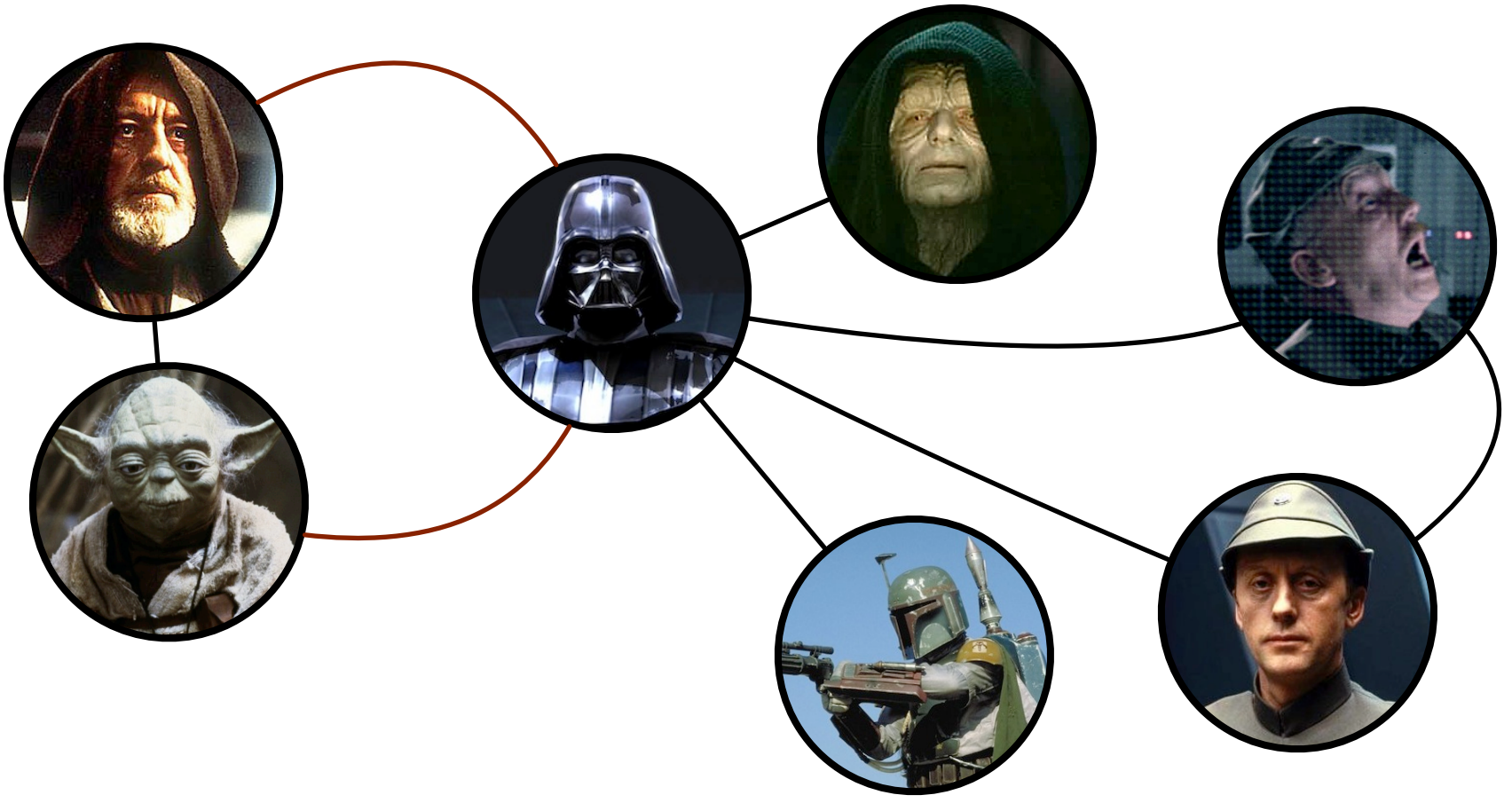
Obi-Wan has a long history with Yoda, and with Darth Vader. They're connected too. Now we are starting to see connections that mean different things. It is complicated, but the connections to Darth Vader indicate opposition.



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 = hostility.

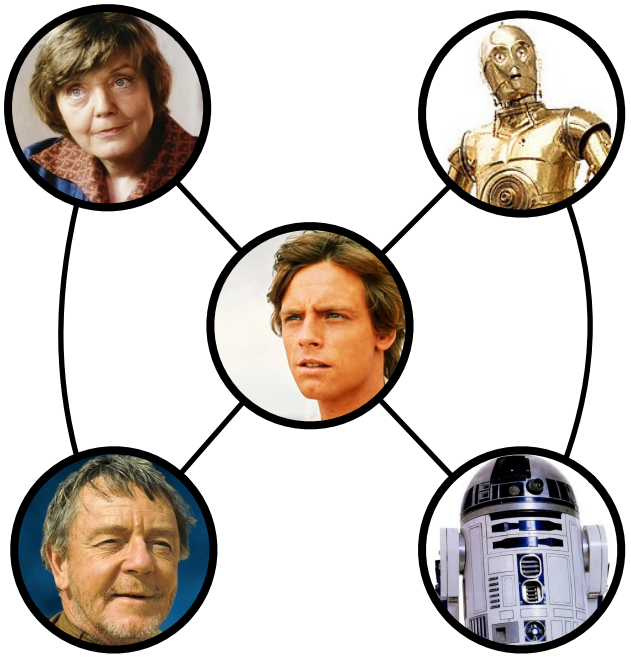
Black = friendship.

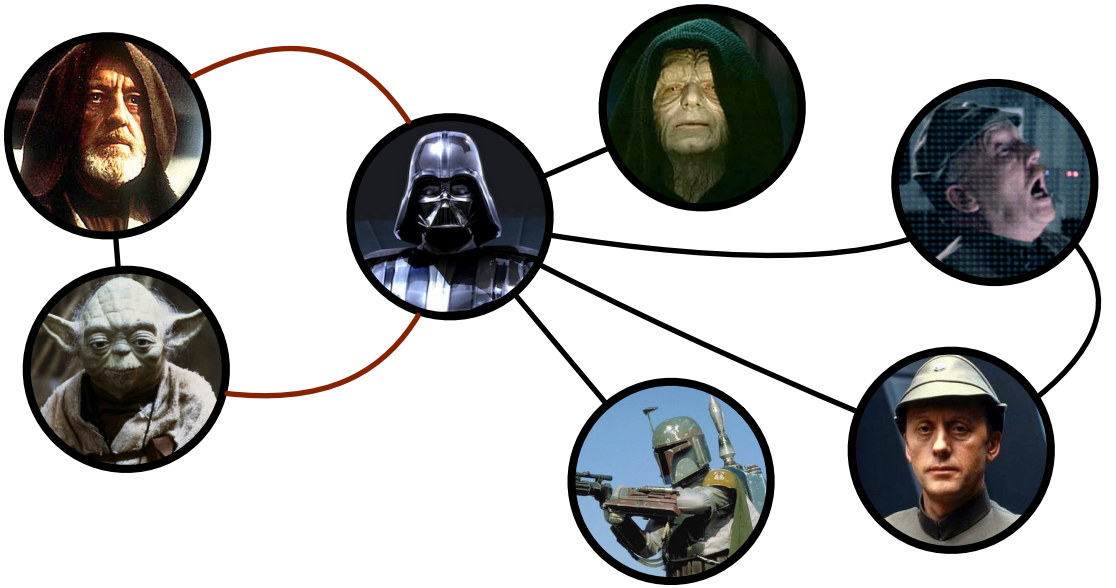


Darth Vader does have some buds, though. The evil Emperor is basically his BFF. Vader worked with these two officers in the Imperial Navy (sadly, the guy at the top had to be *dealt with*). And then there's good ol' Boba Fett, bounty hunter.

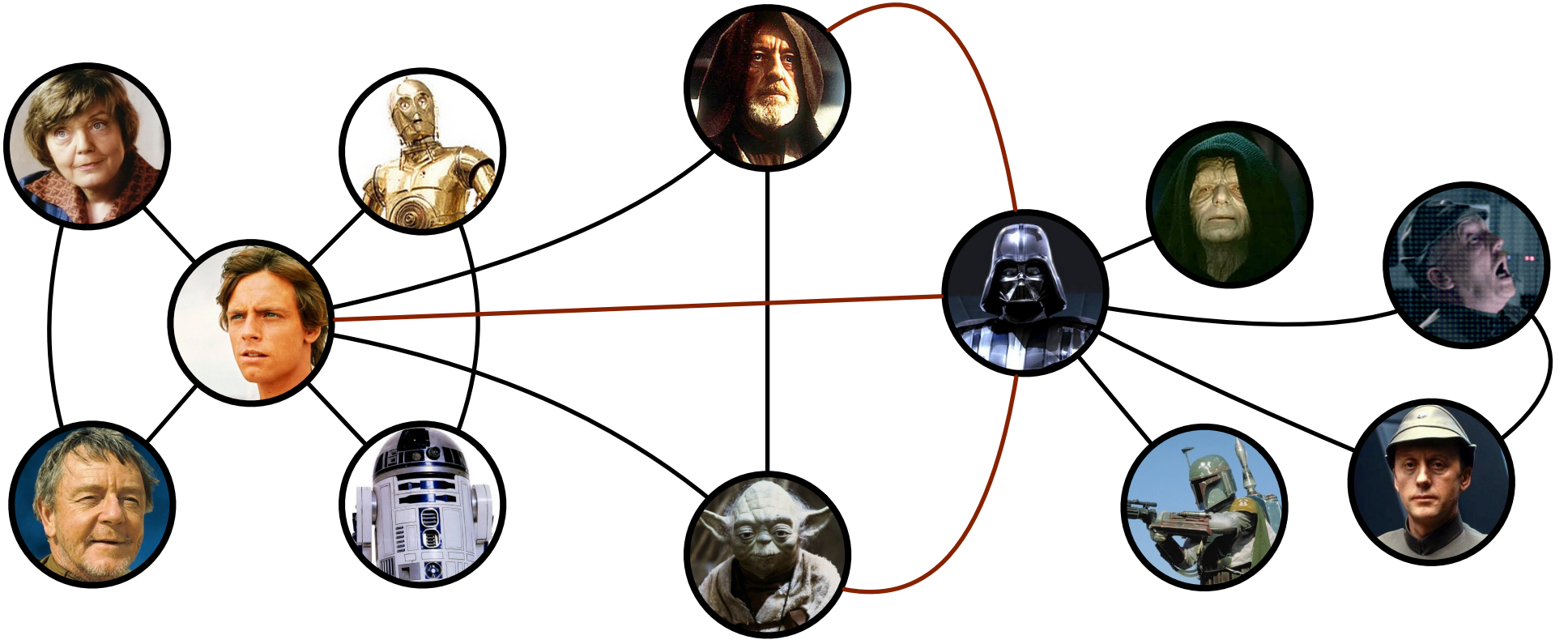
Red = enemies.
Black = allies.

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

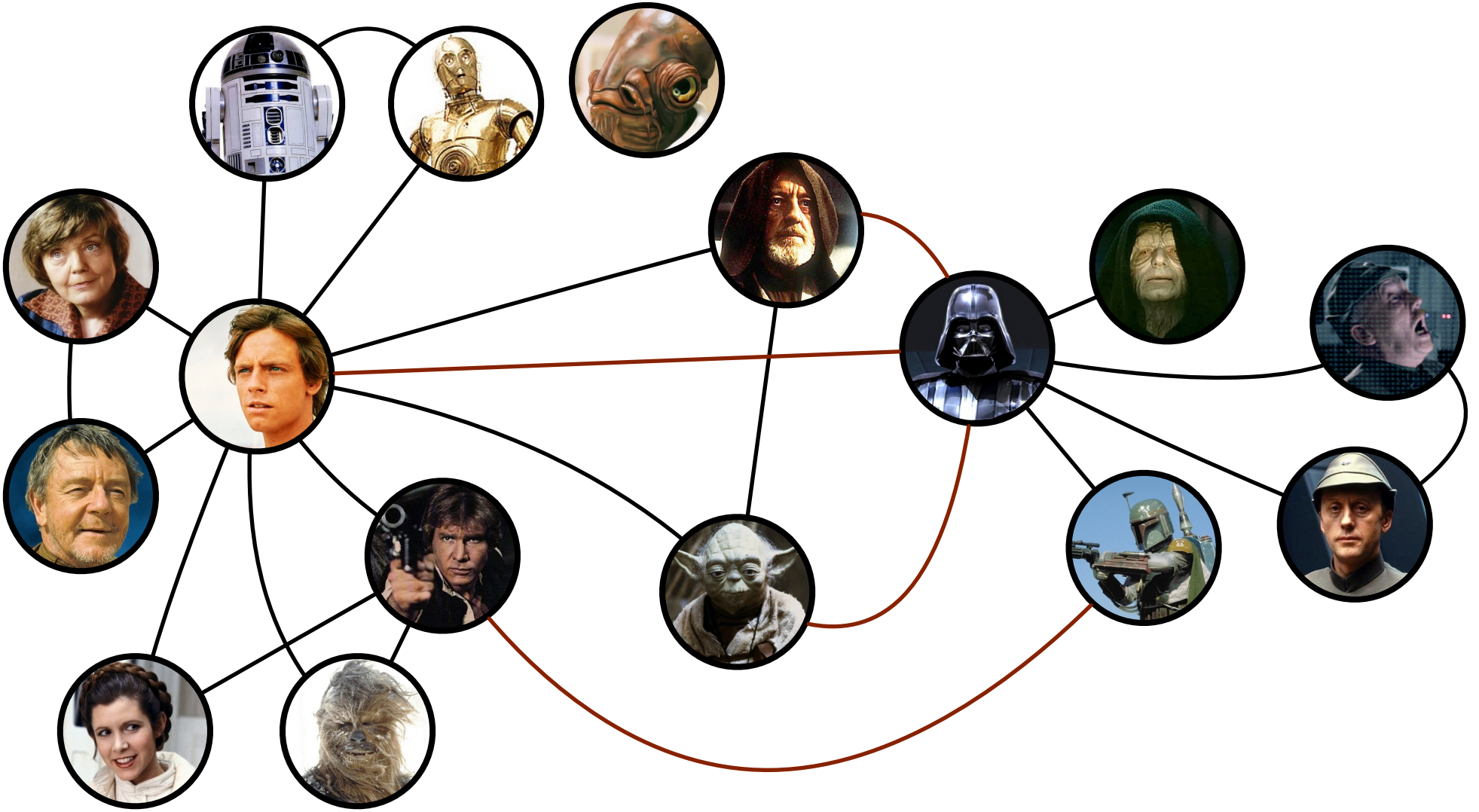




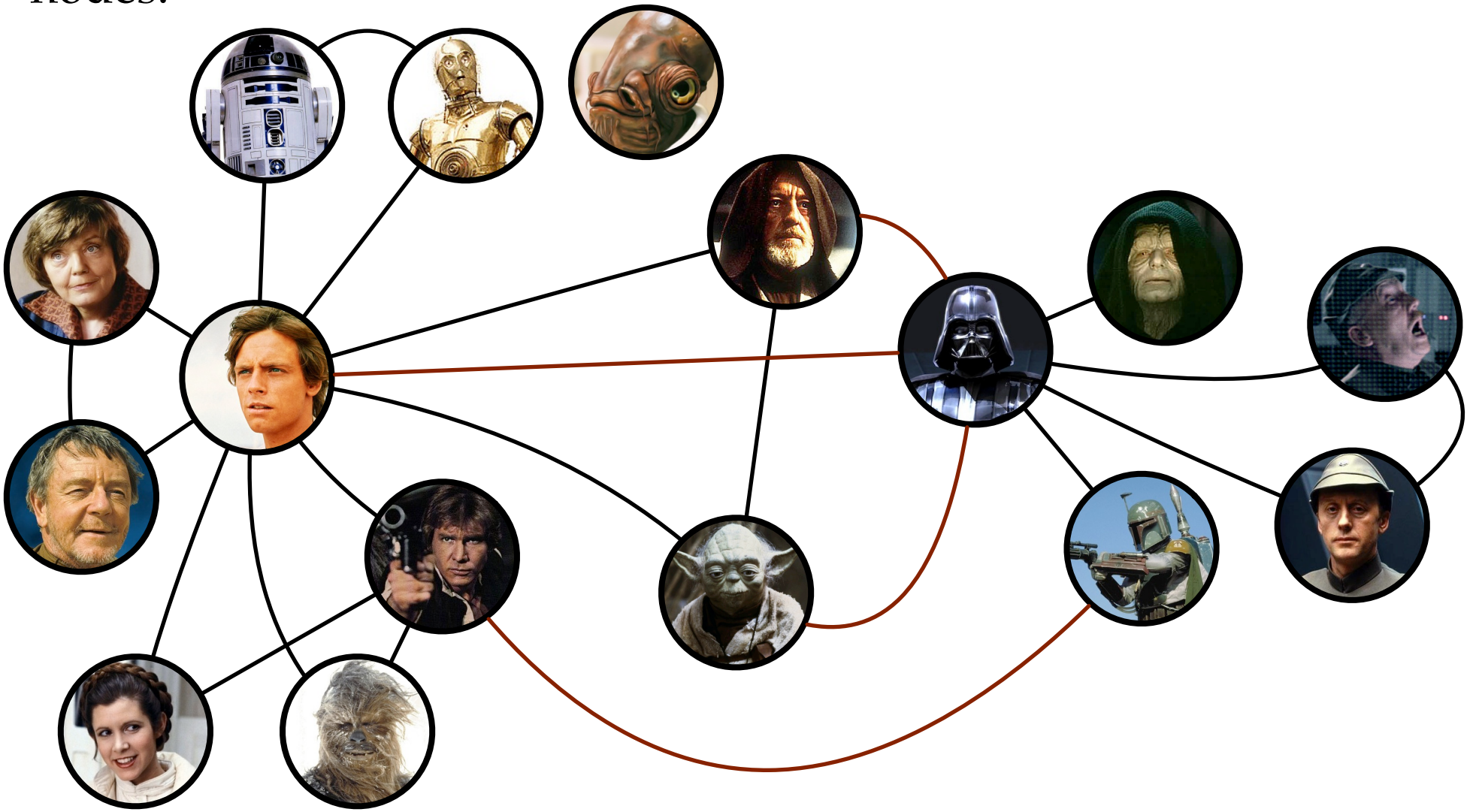
Luke meets up with Obi-Wan and Yoda, and joins the battle vs. Darth Vader.



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



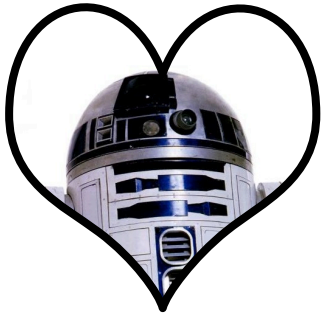
Notice how poor Admiral Ackbar is in the graph, but he's not *connected* to anyone. Aside from being kind of sad, this is actually a legitimate thing. Graphs nodes don't have to be connected to other nodes.



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.

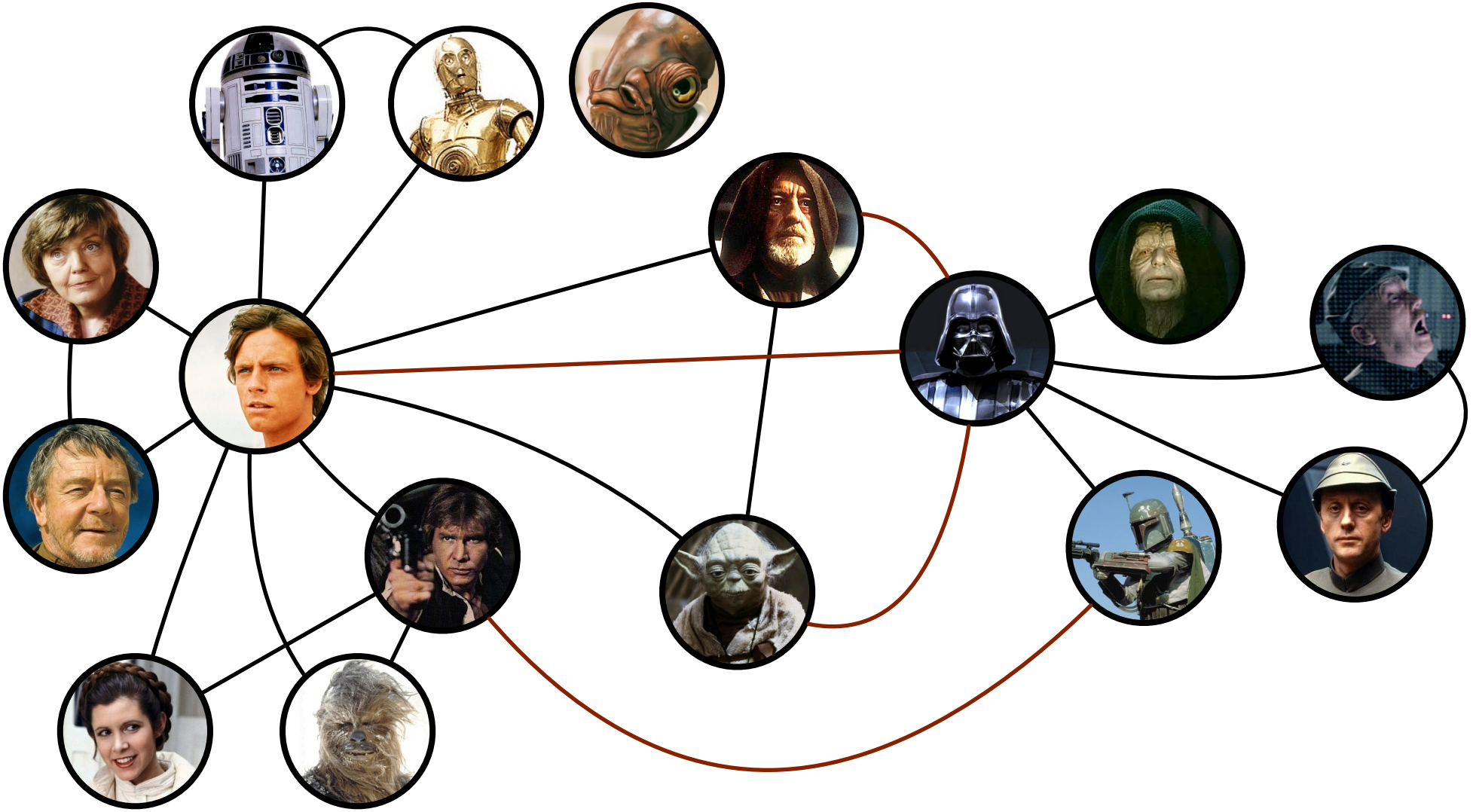
How do we do this?  = { ? }

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.

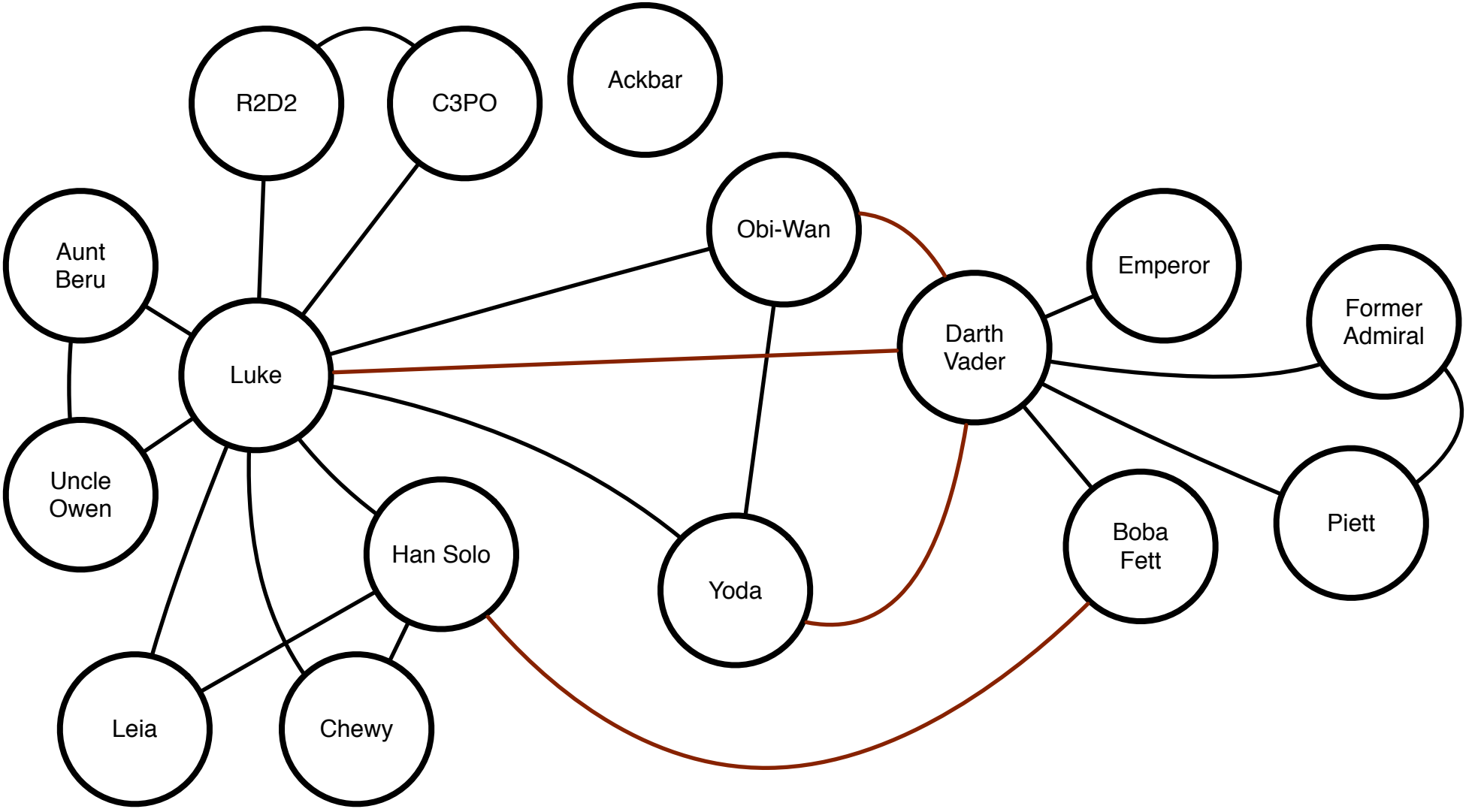
The basic **Depth-First Search** algorithm is: pop a node from the stack, color it gray, visit it, iterate through neighboring nodes, and run the DFS process on each in turn. When leaving, color node black.

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

Here's the whole graph:

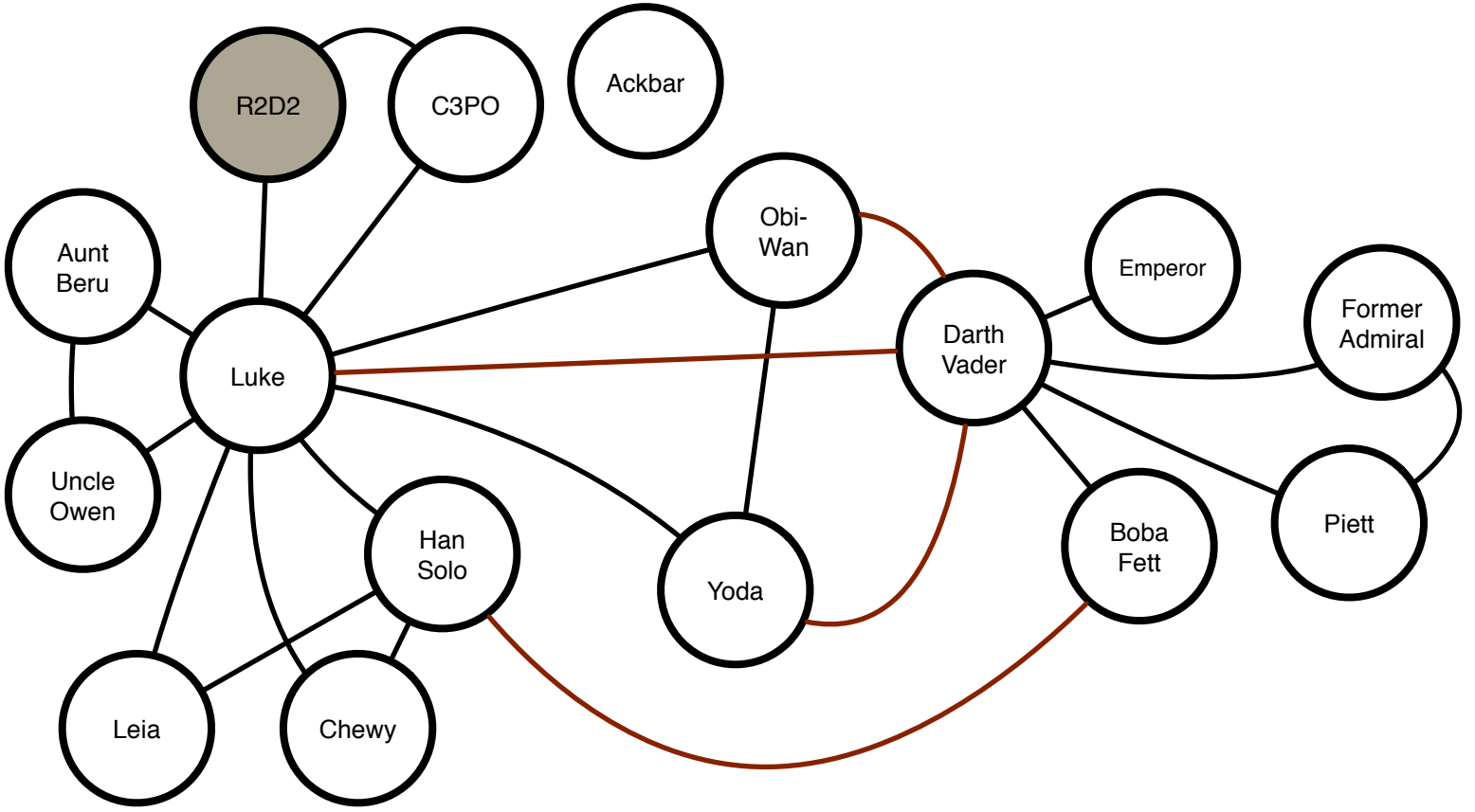


We'll start from R2D2. But I'll replace the pictures with labels.



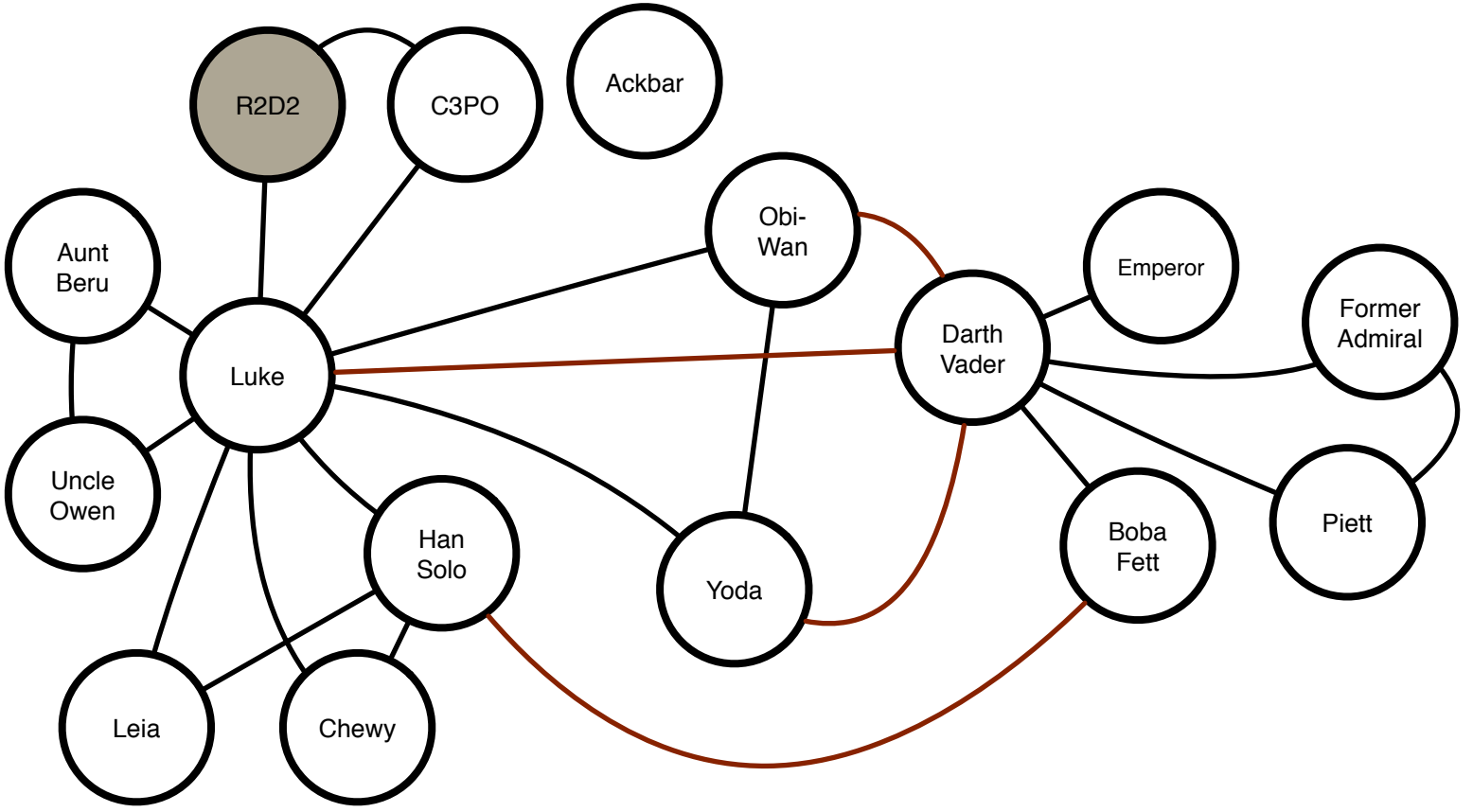
First thing to do when we begin a node is to mark it as 'in progress'. Often this is called *coloring* a node.

`White = Unexplored; Gray = In Progress; Black = Done.`



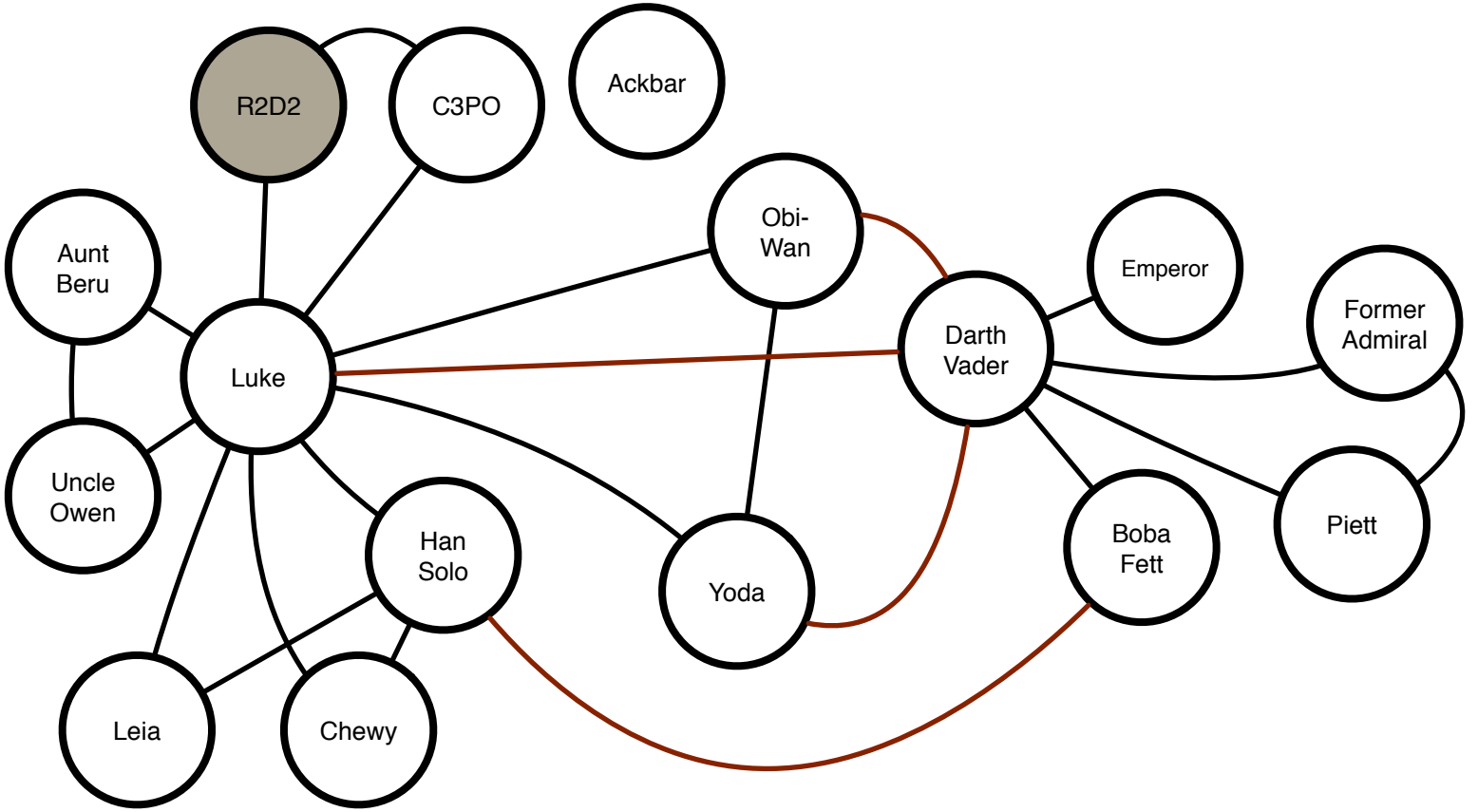
Next: assemble a list of nodes that are adjacent to our node, using only 'friendly' edges. **C3PO** and **Luke** are next to R2D2. Keep neighboring nodes in a data structure. A *stack* is common.

R2D2: C3PO, Luke



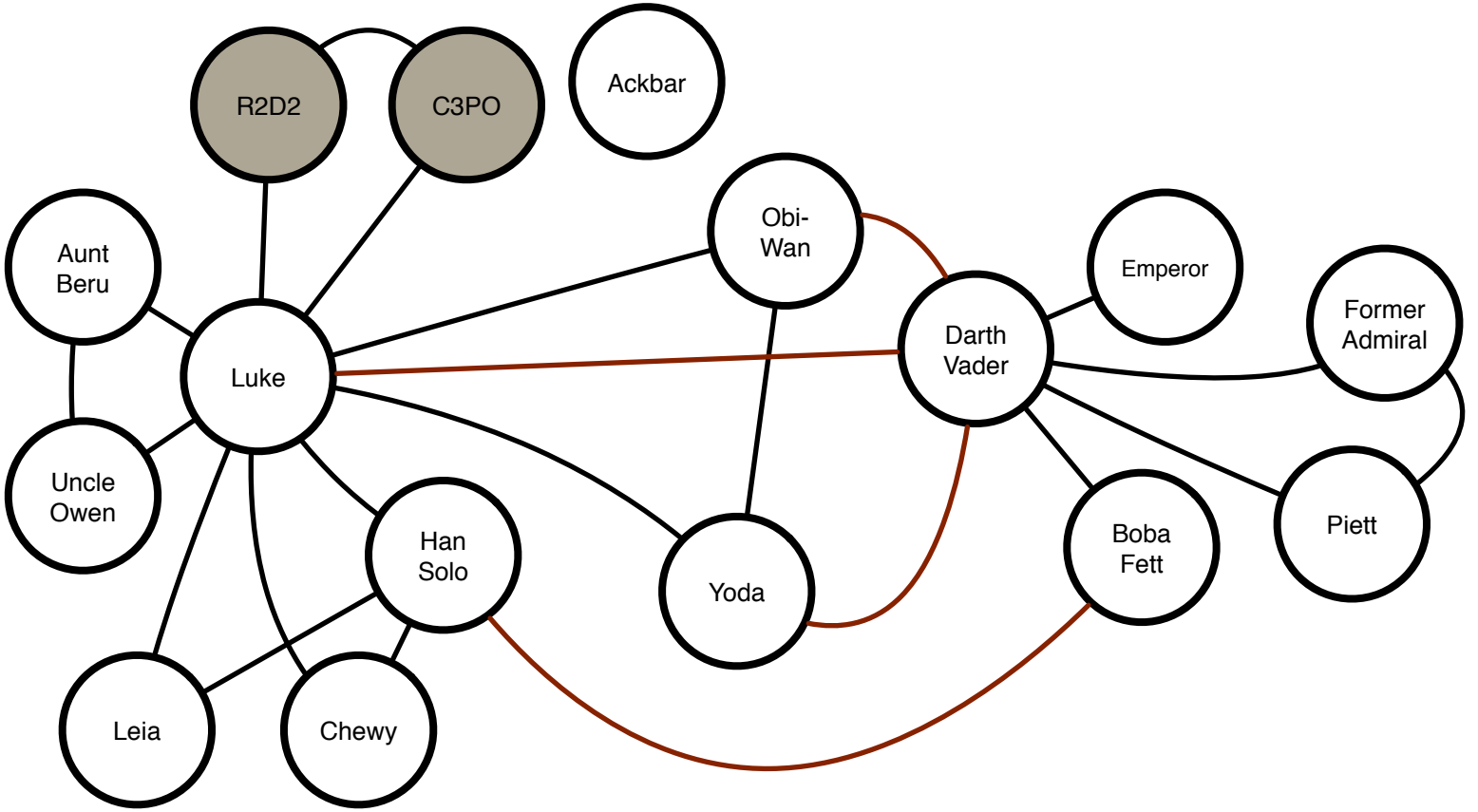
Now, pop one node from the current node's stack, and visit it. 'C3PO' is the next node, so that's where we'll go.

R2D2: **C3PO**, Luke



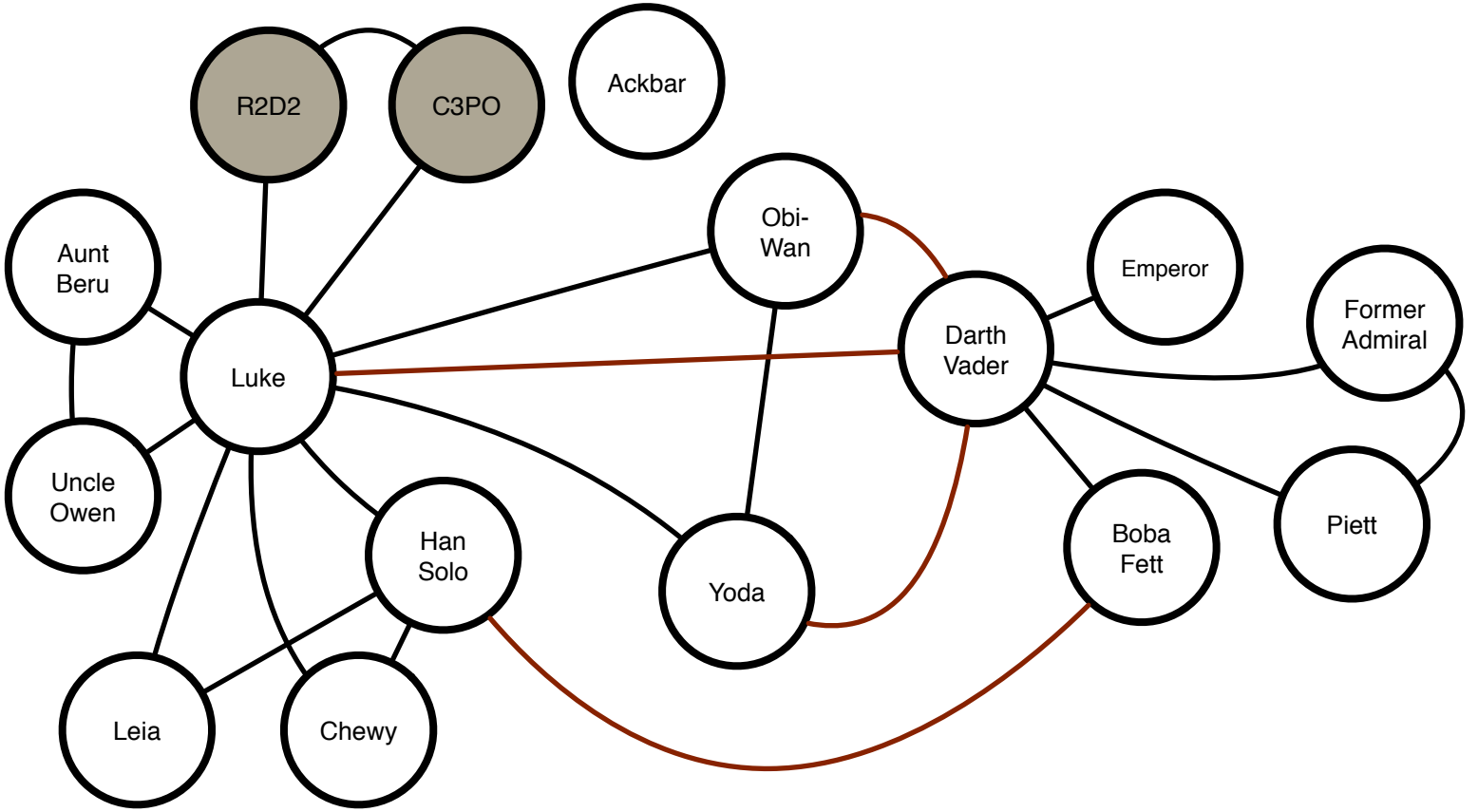
Mark C3PO as 'In Progress' (Gray), and form it's list of adjacencies. Notice that R2D2 is on the list!

C3PO: R2D2, Luke
R2D2: Luke



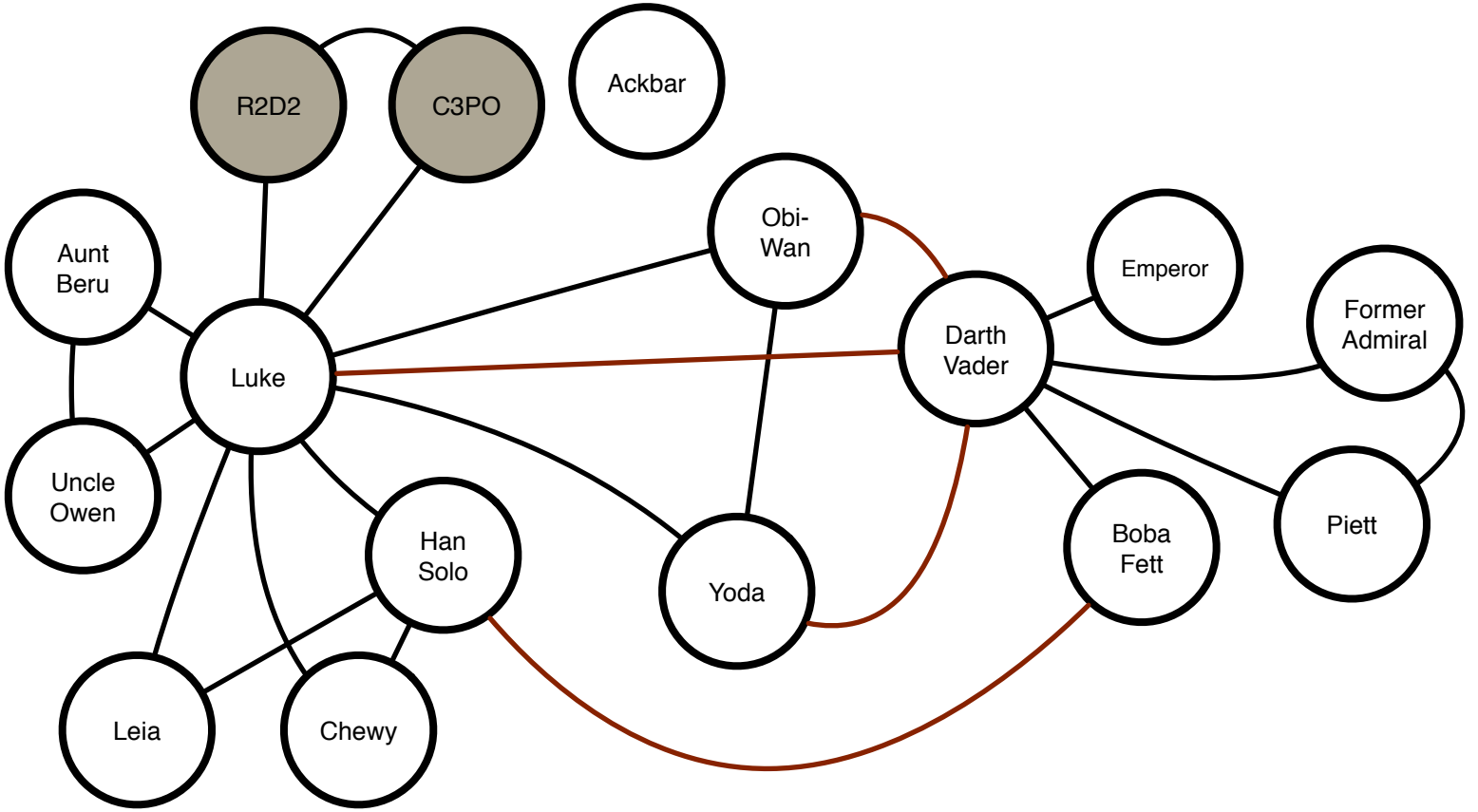
Pop from C3PO's list to find R2D2. But R2D2 is currently in progress, so we will NOT follow that link.

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



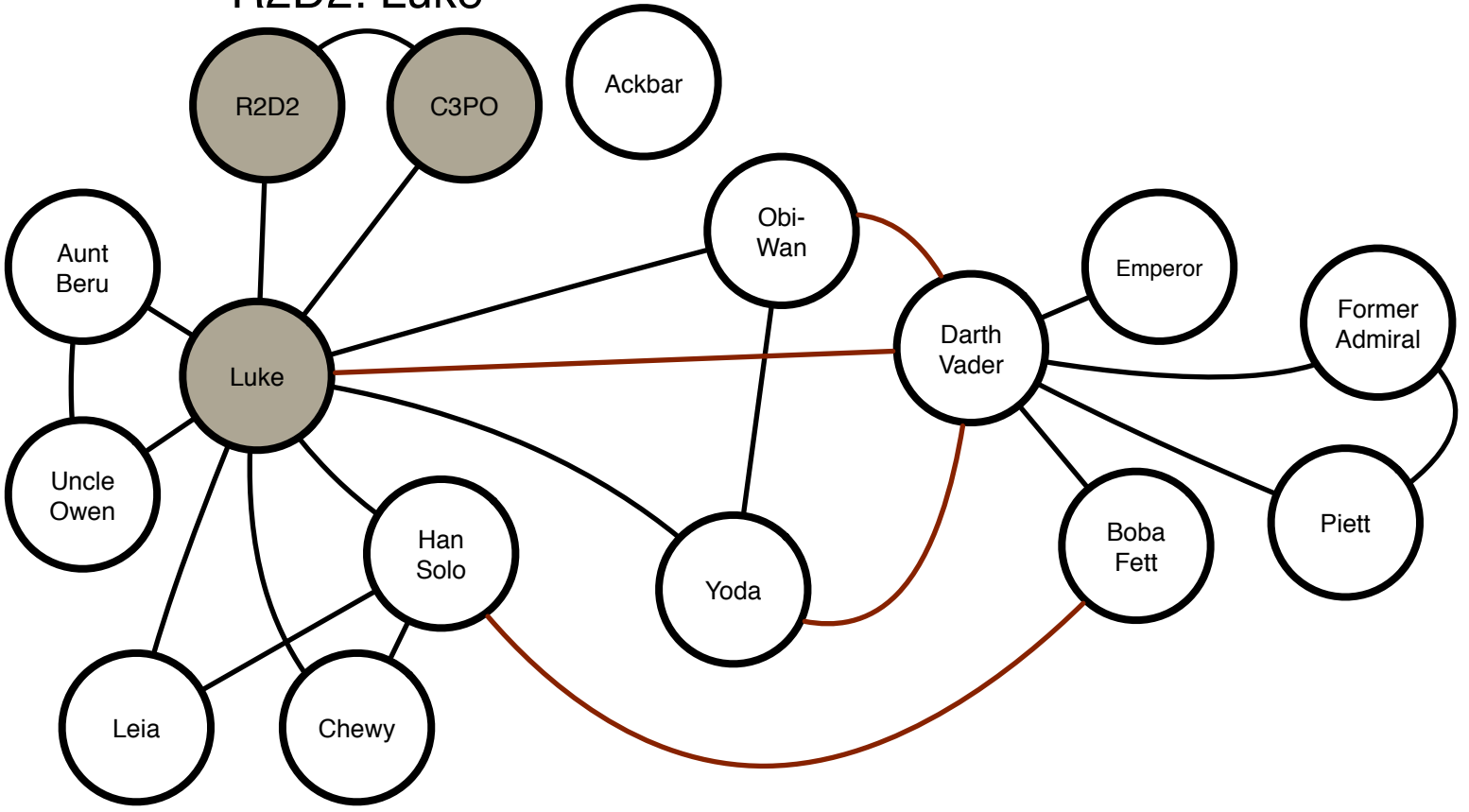
Pop the next thing from C3PO's stack to get Luke.

C3PO: **Luke**
R2D2: Luke



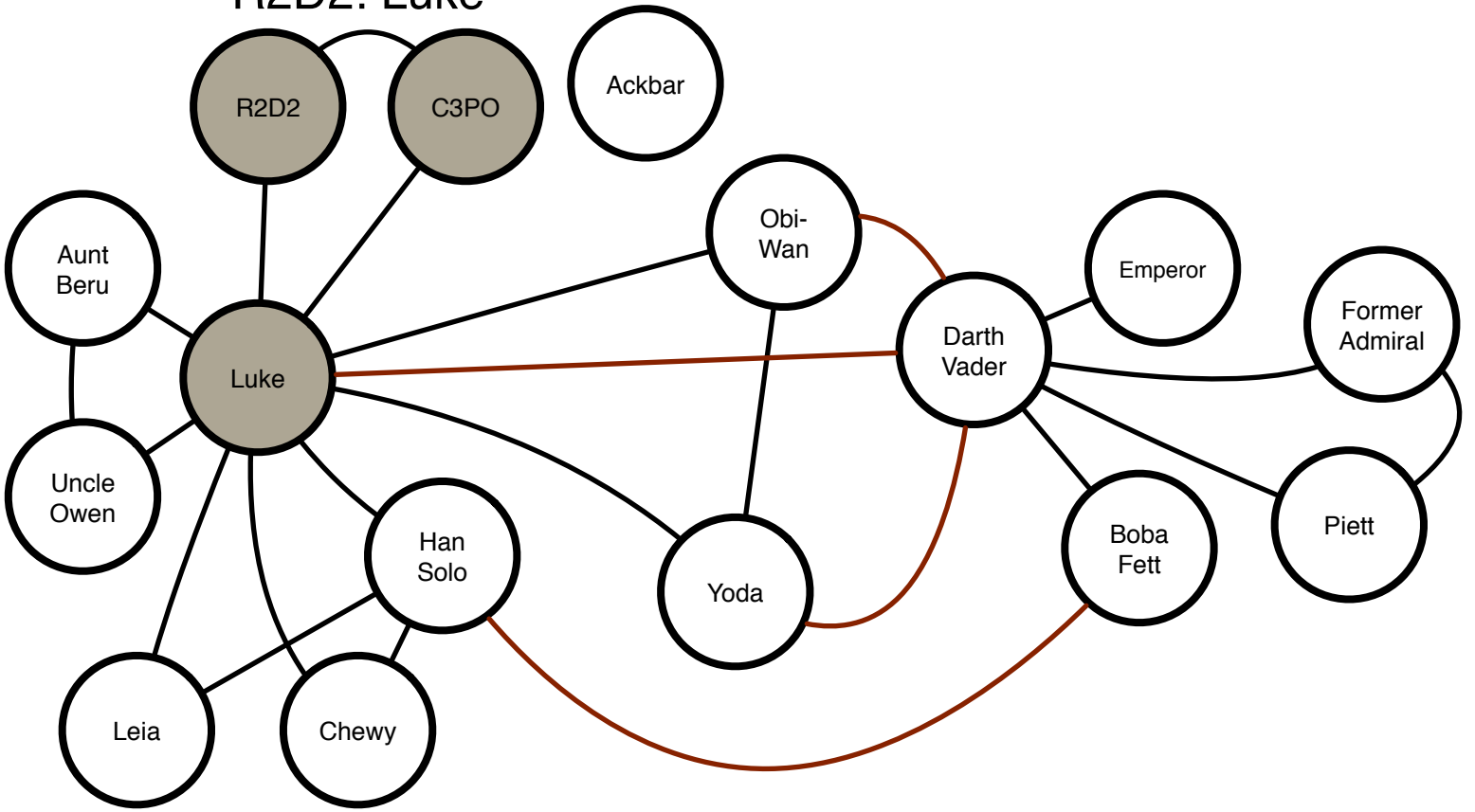
(Notice that C3PO's stack is now empty.)
Now we begin visiting Luke. Mark him Gray, form a list of adjacent nodes.

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



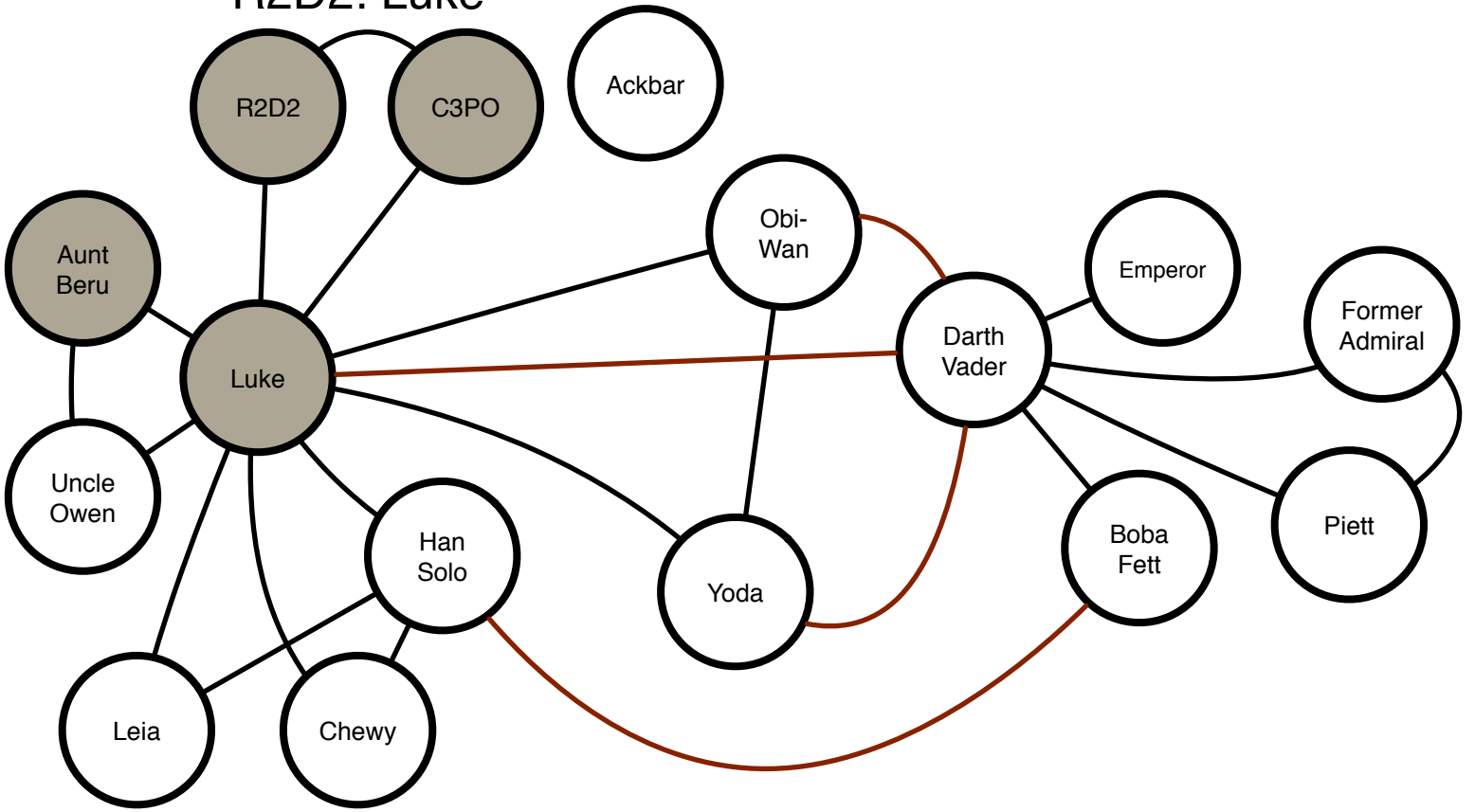
Pop from Luke's stack to get Aunt Beru. That node is White, so it is fair game.

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



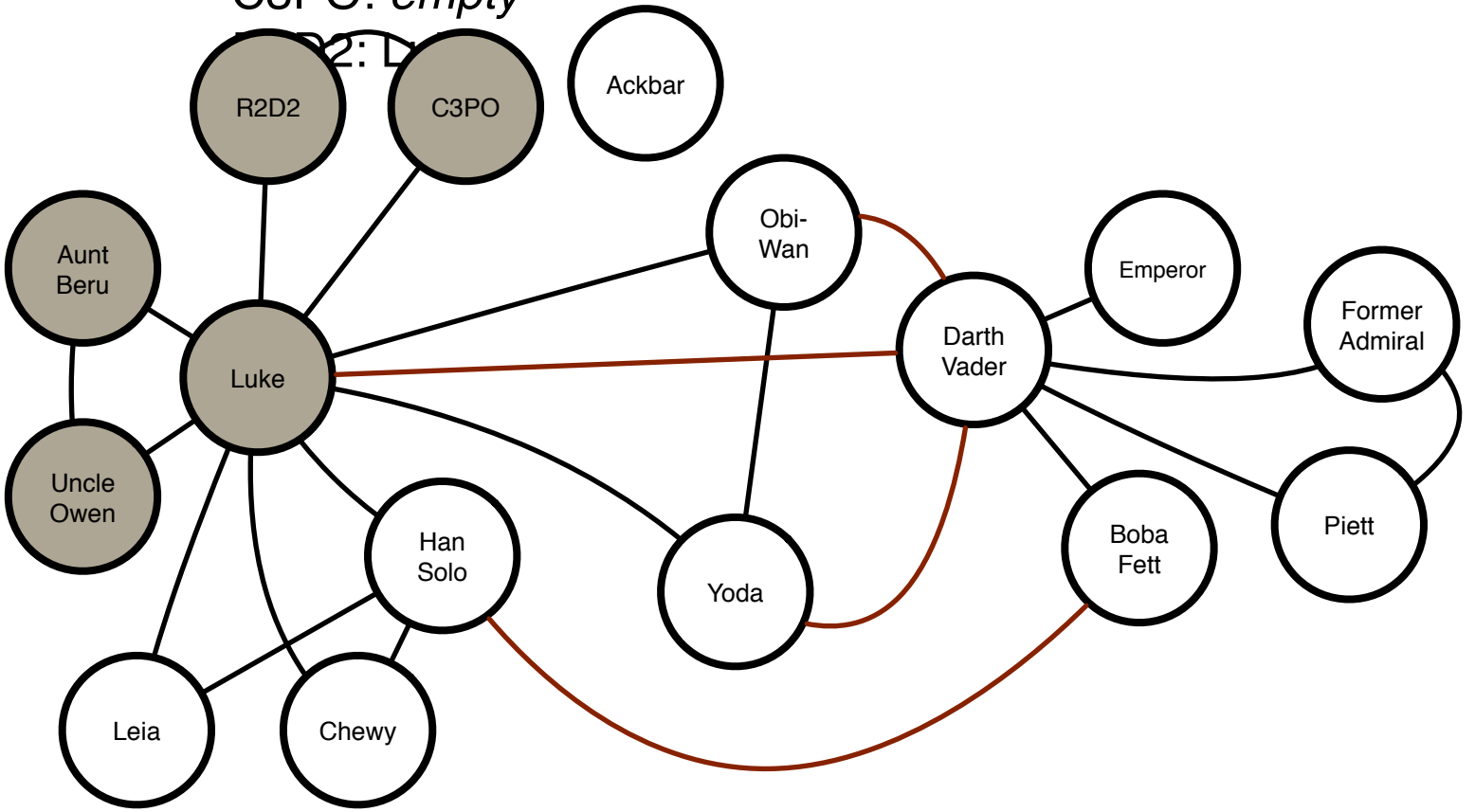
Visiting Aunt Beru. First item in her stack is Luke, but after removing him from the stack we see he's Gray. So we continue on to Uncle Owen. Beru's stack becomes empty.

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



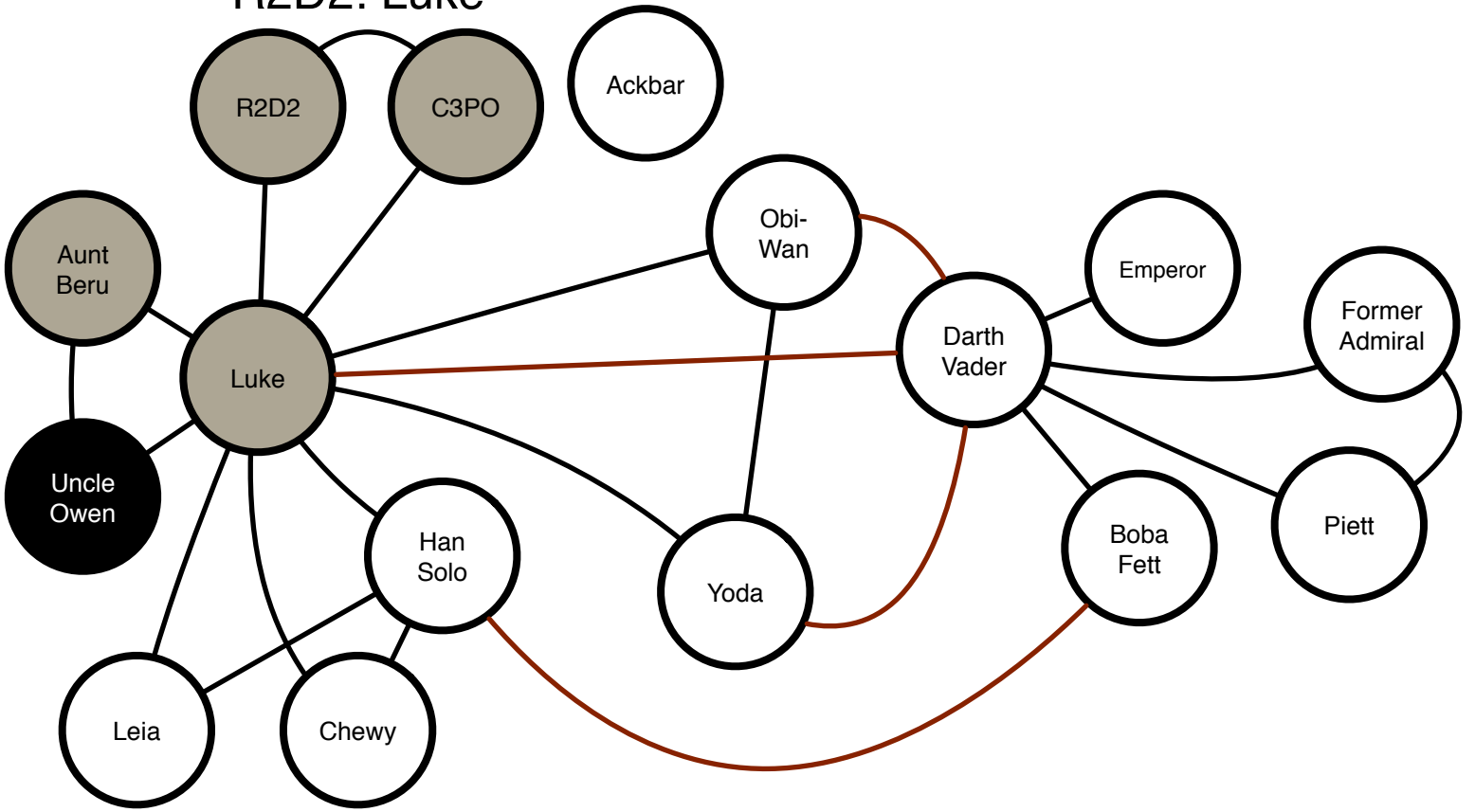
Visiting Uncle Owen. Make a stack with his relations: Beru and Luke. But after popping each from the stack we see they are Gray already. This means Owen is fully complete. Mark it Black.

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



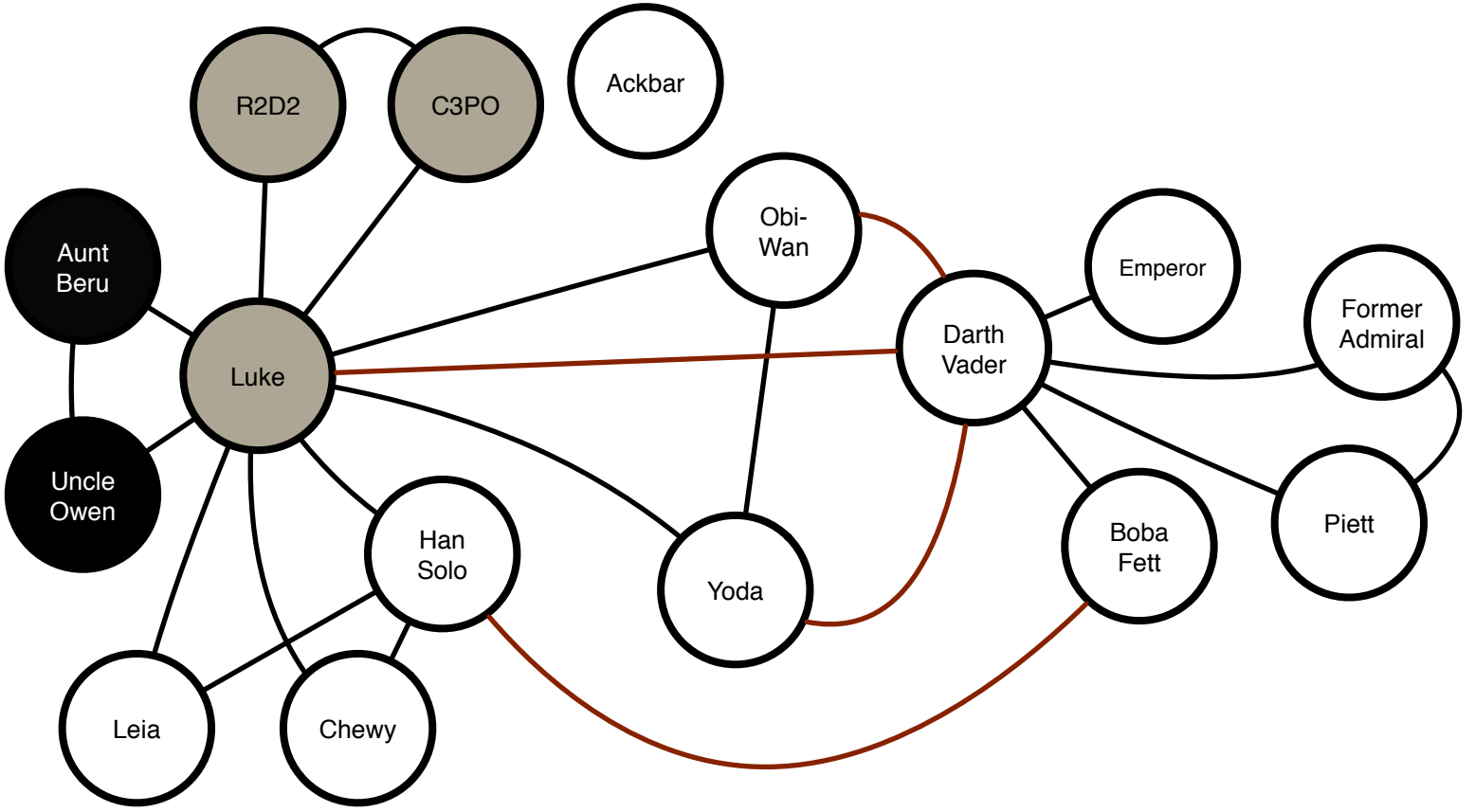
Owen is done, so we return from whence we came: Aunt Beru. Her stack is *also* empty. So, mark that node Black as well.

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



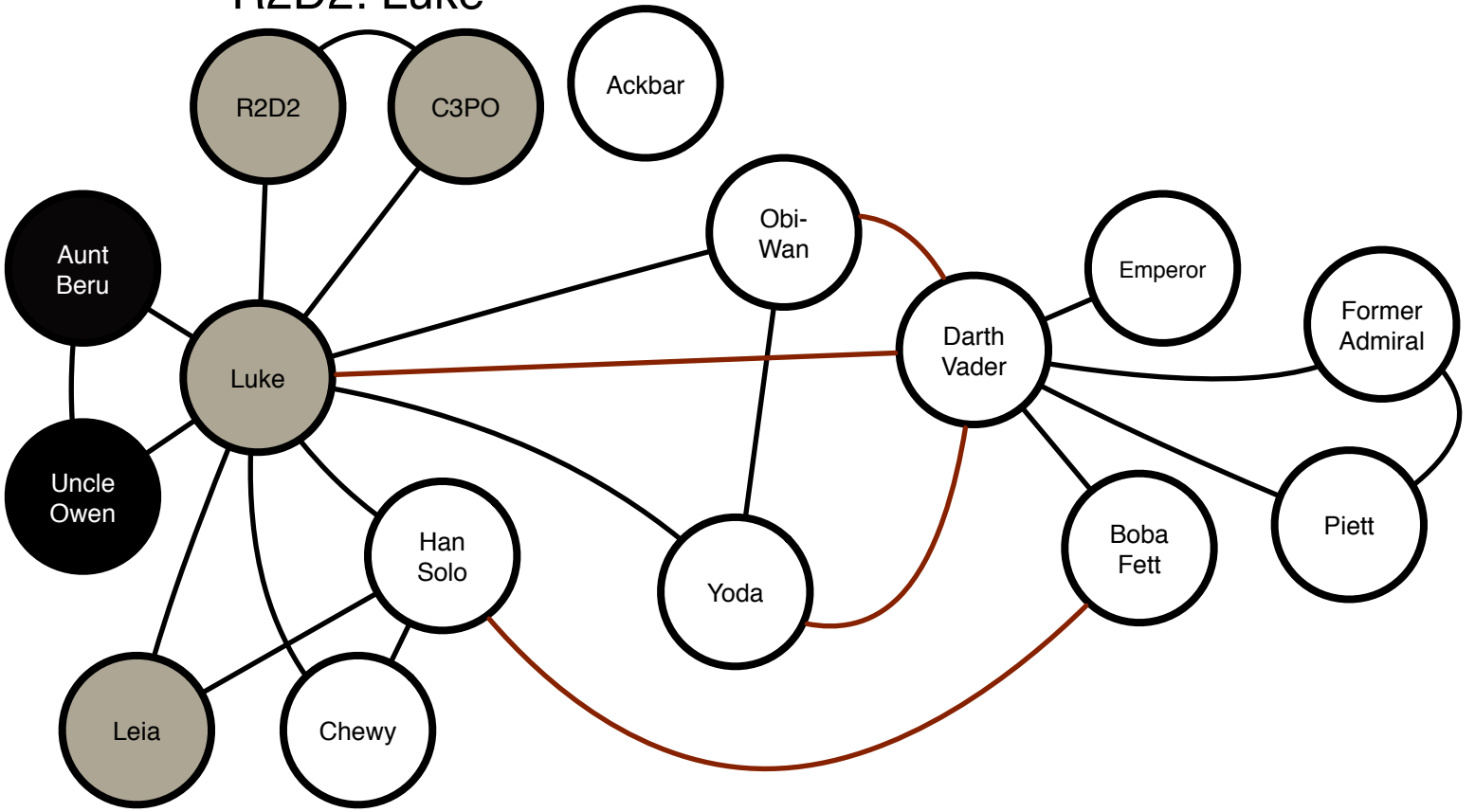
Back at Luke now. Notice how we are diving into the graph and going as far as we can before turning nodes black. That's how a DFS rolls. Anyway: Uncle Owen is Black, so explore to Leia.

Luke: Uncle Owen, **Leia**, Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
R2D2: Luke



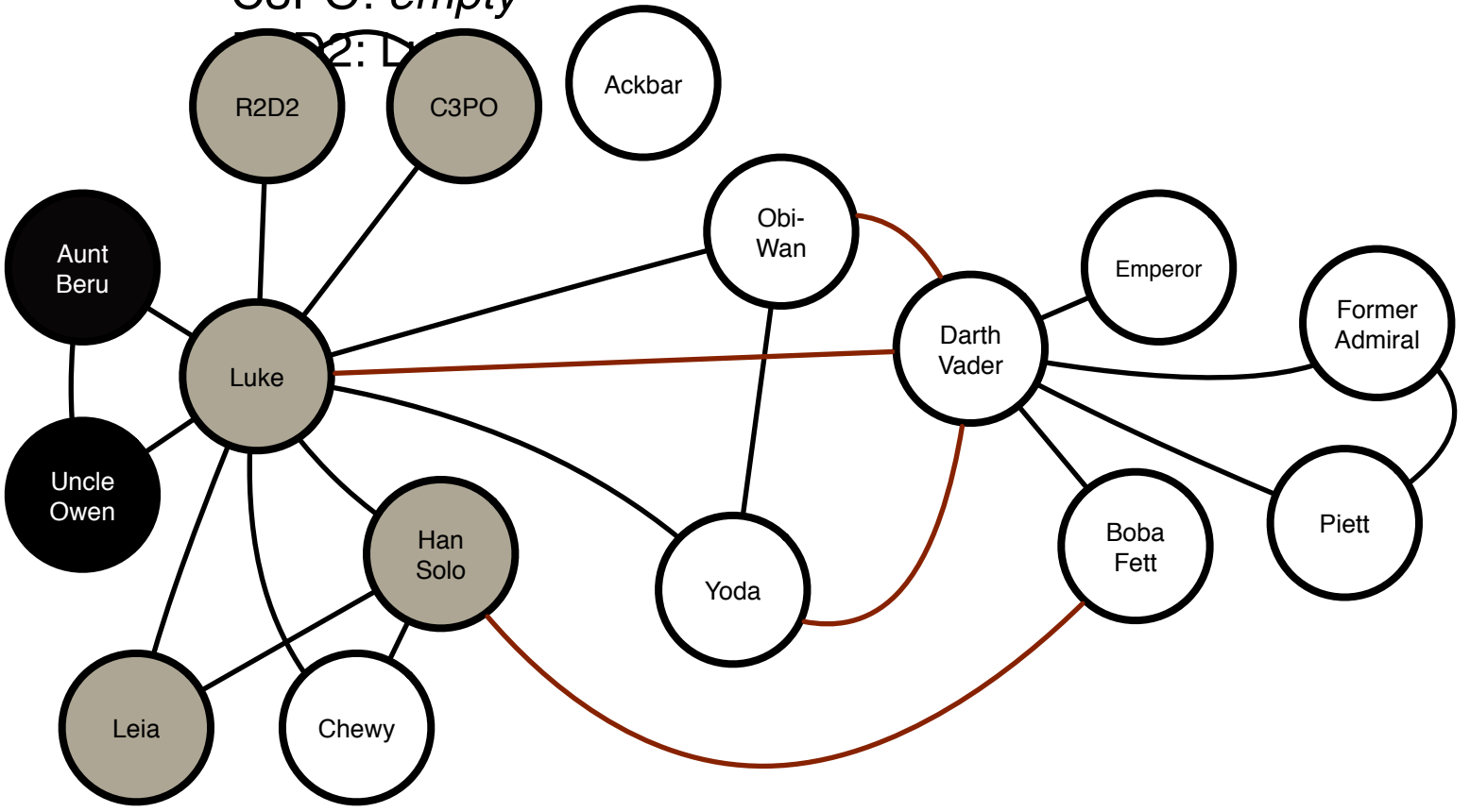
Leia's stack is Luke and Han Solo. Luke is Gray, so ignore him. Move on to Han Solo.

Leia: Luke, **Han Solo**
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
R2D2: Luke

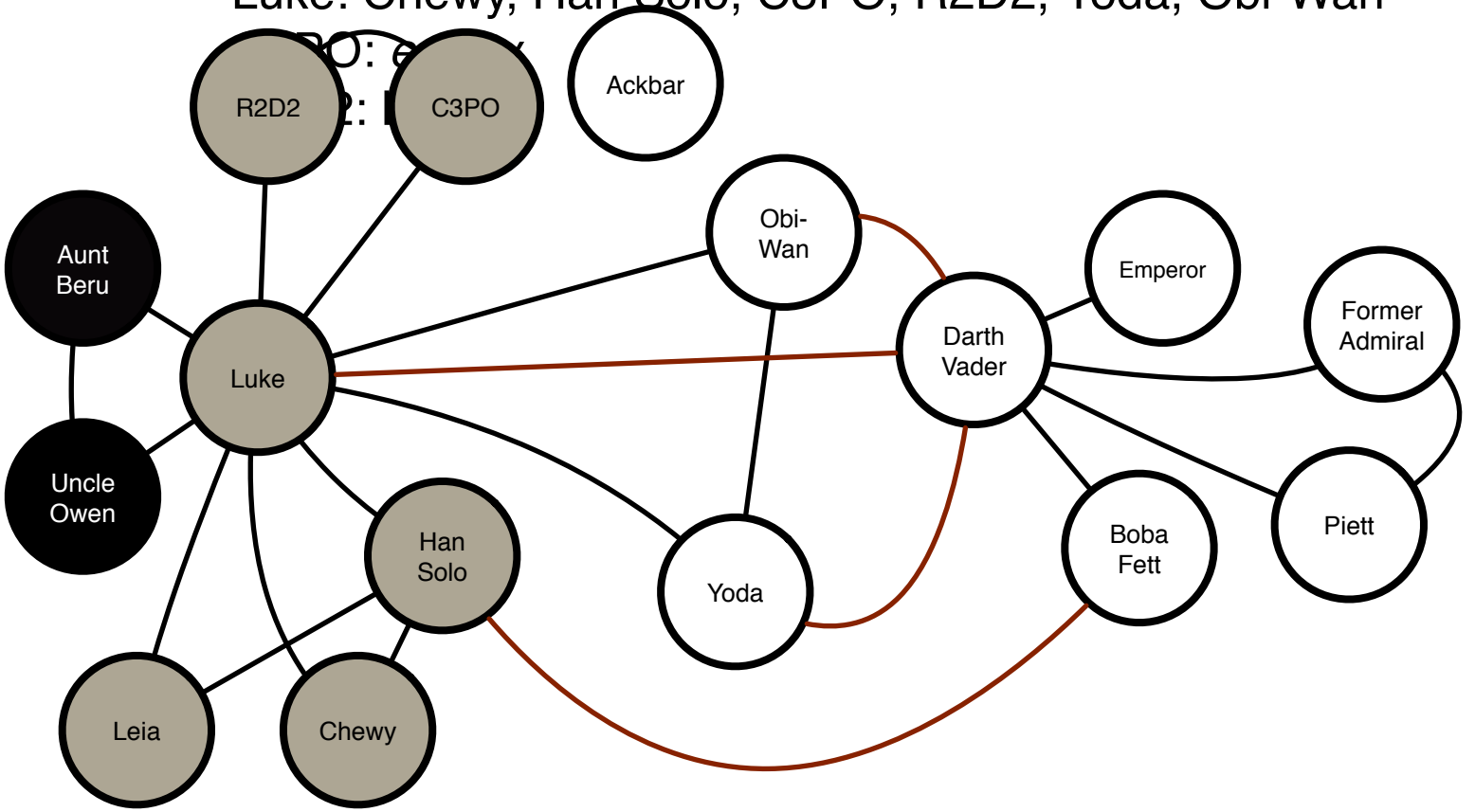


I'm going to stop giving the play-by-play since the diagram should by now speak for what's up.

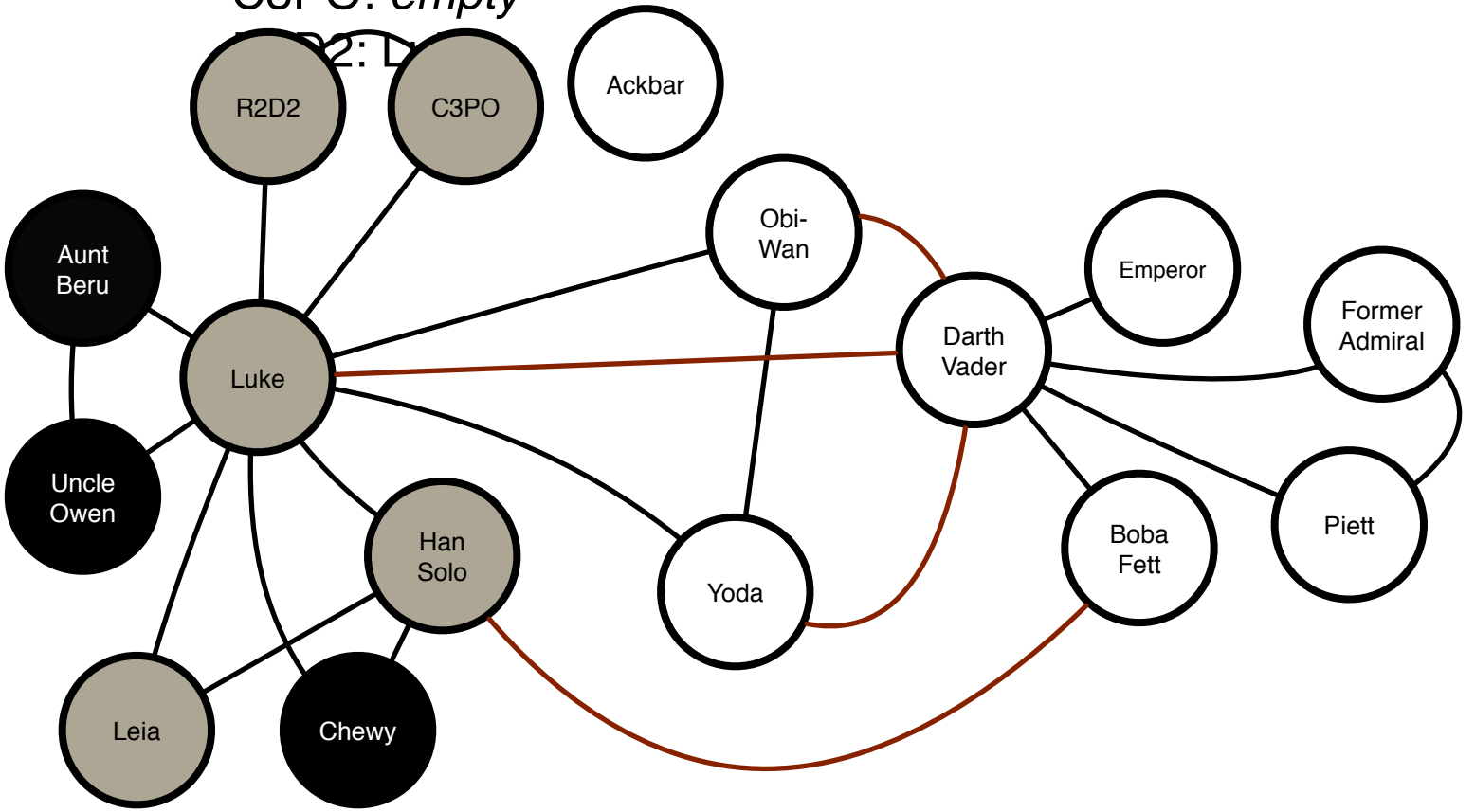
Han Solo: Luke, Chewy
Leia: *empty*
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
[illegible]



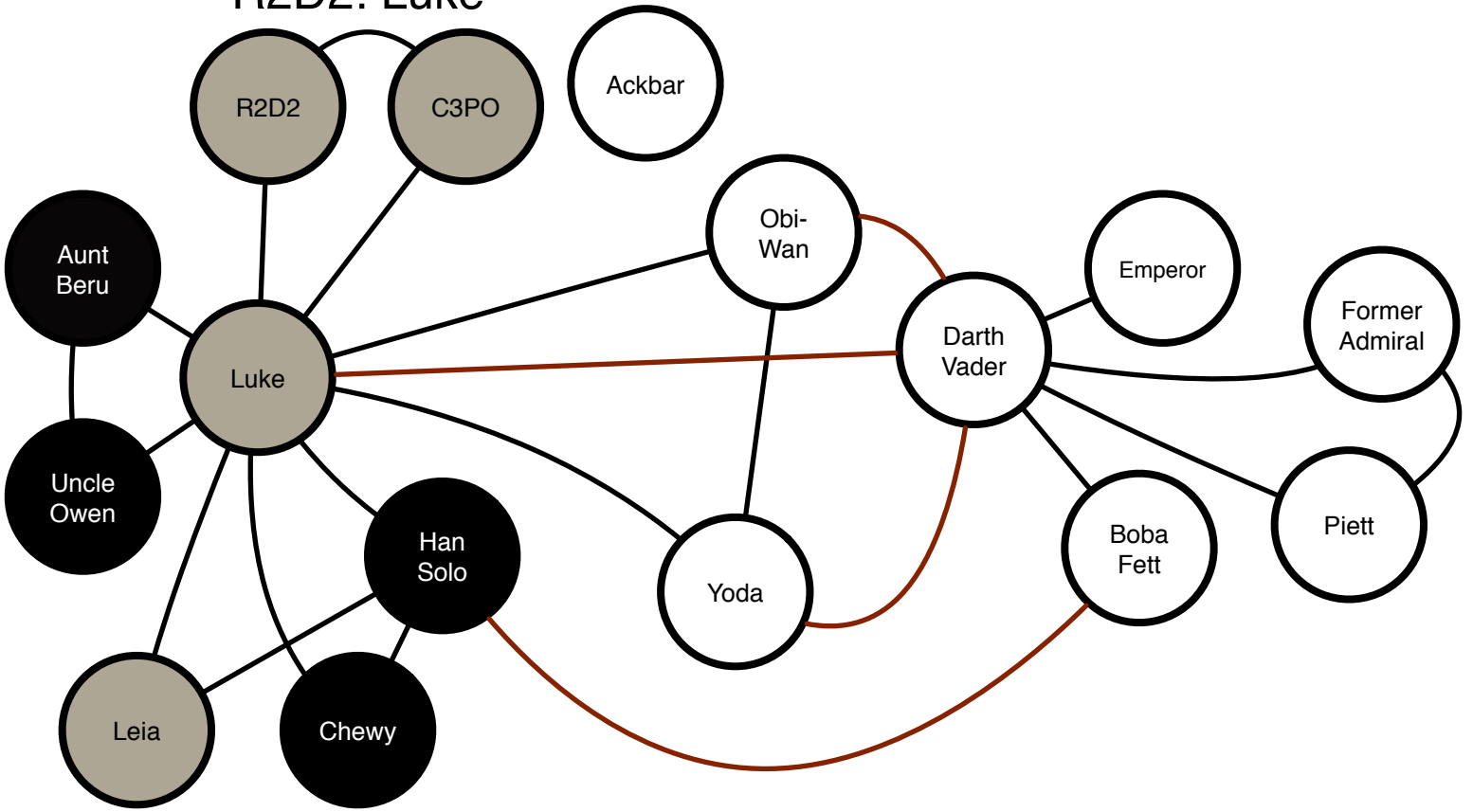
Chewy: Luke, Han Solo
Han Solo: *empty*
Leia: *empty*
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan



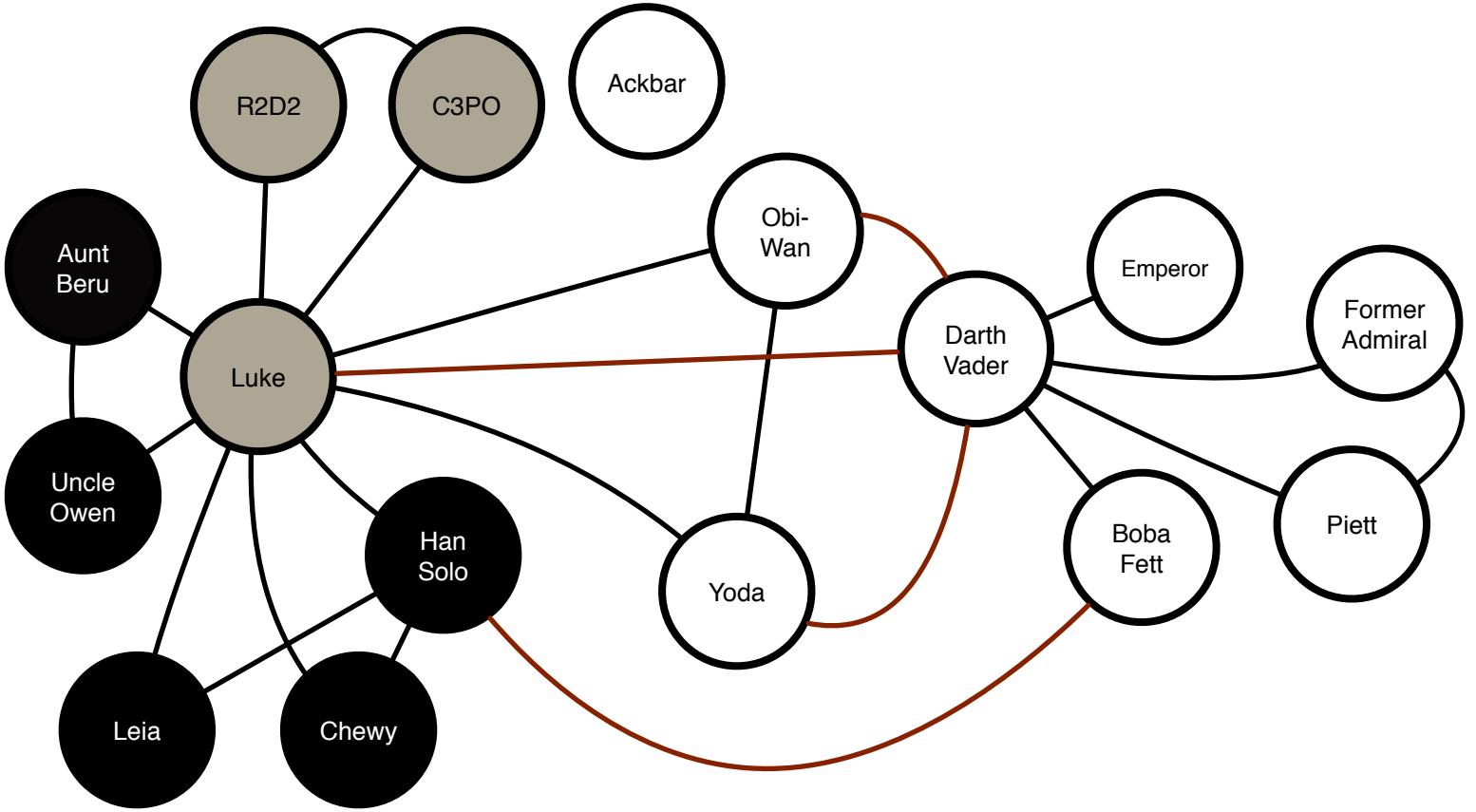
Han Solo: *empty*
Leia: *empty*
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
R2D2: L



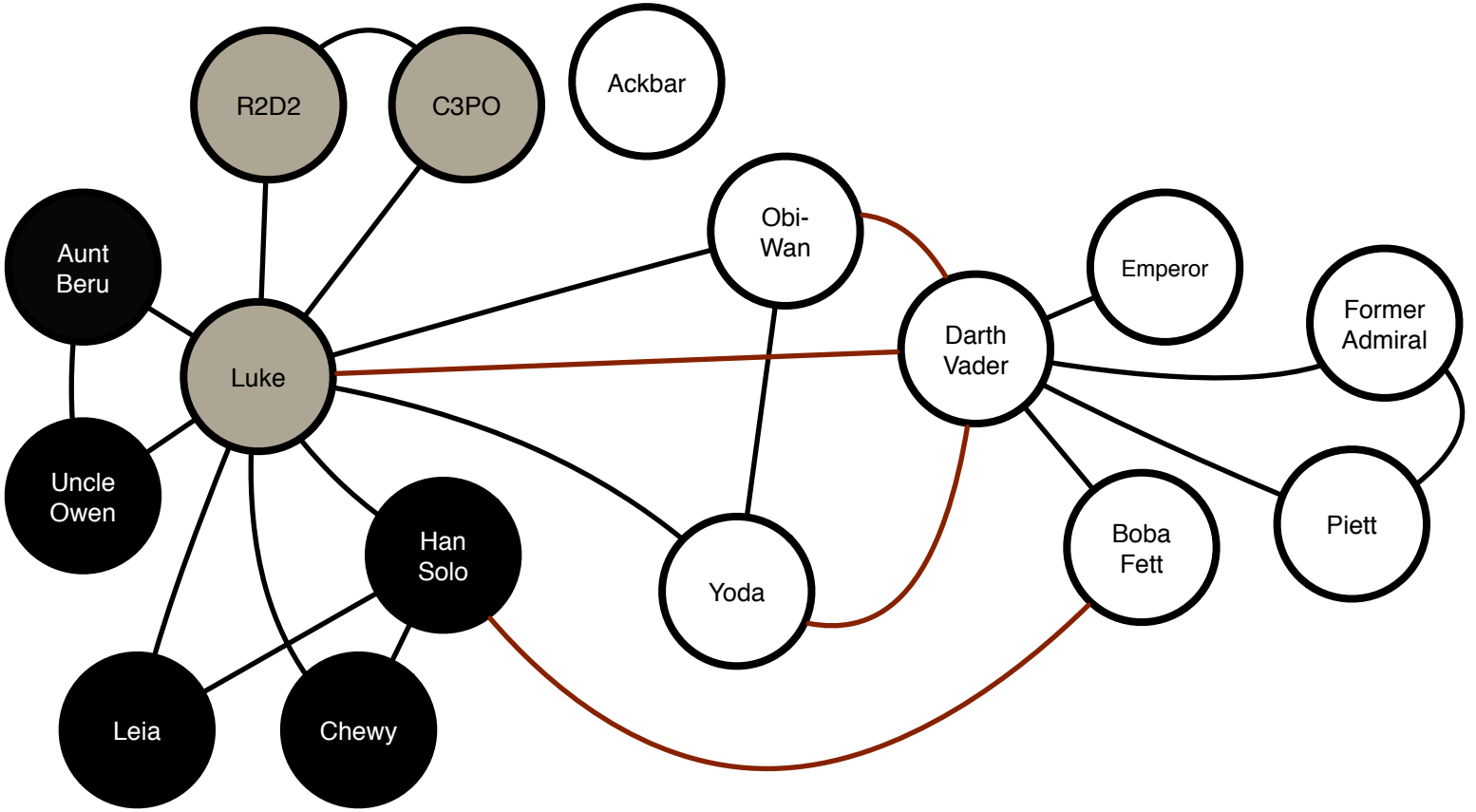
Leia: *empty*
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
R2D2: Luke



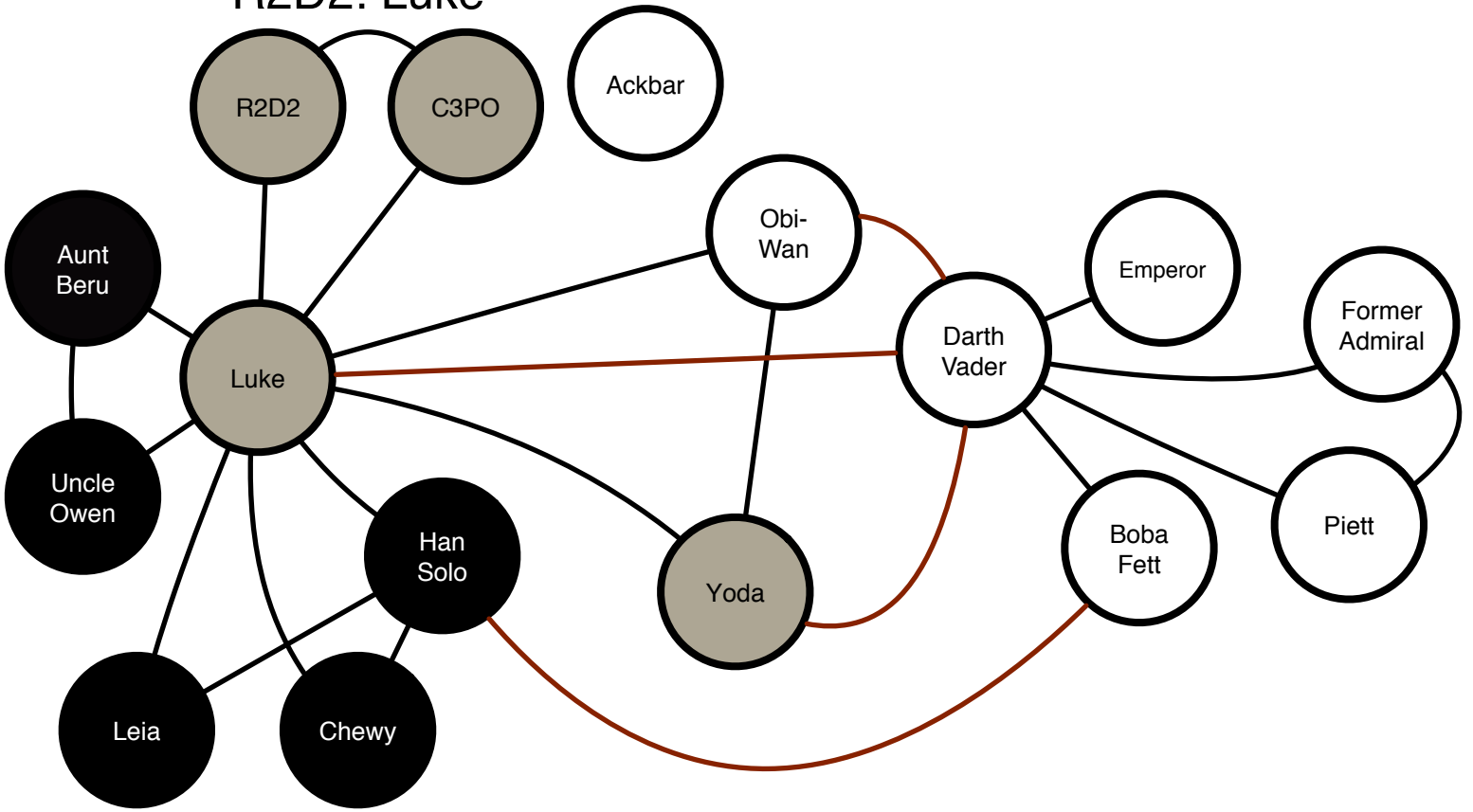
Luke: Chewy, Han Solo, C3PO, R2D2, Yoda, Obi-Wan
C3PO: *empty*
R2D2: Luke



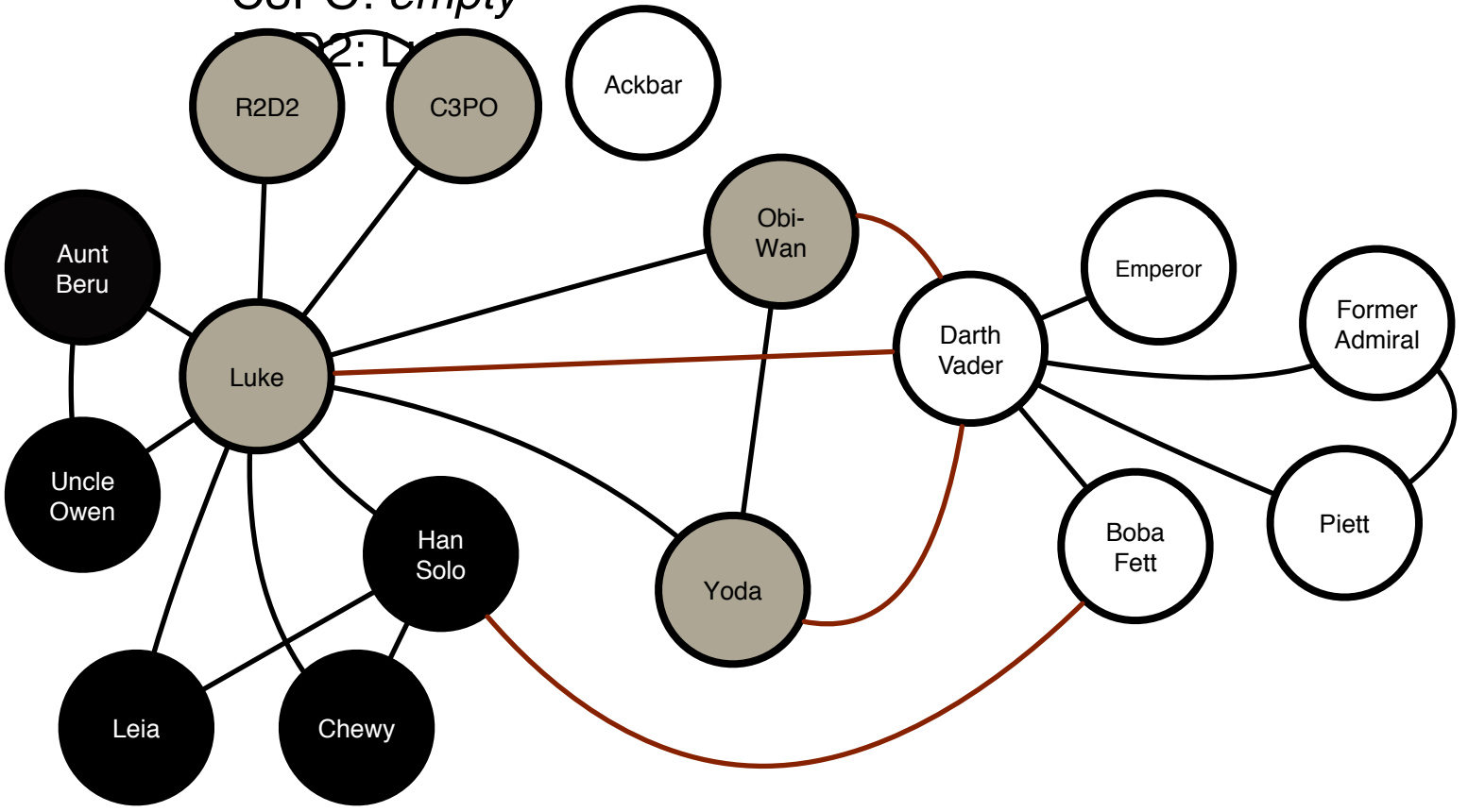
Luke: **Yoda**, Obi-Wan
C3PO: *empty*
R2D2: Luke



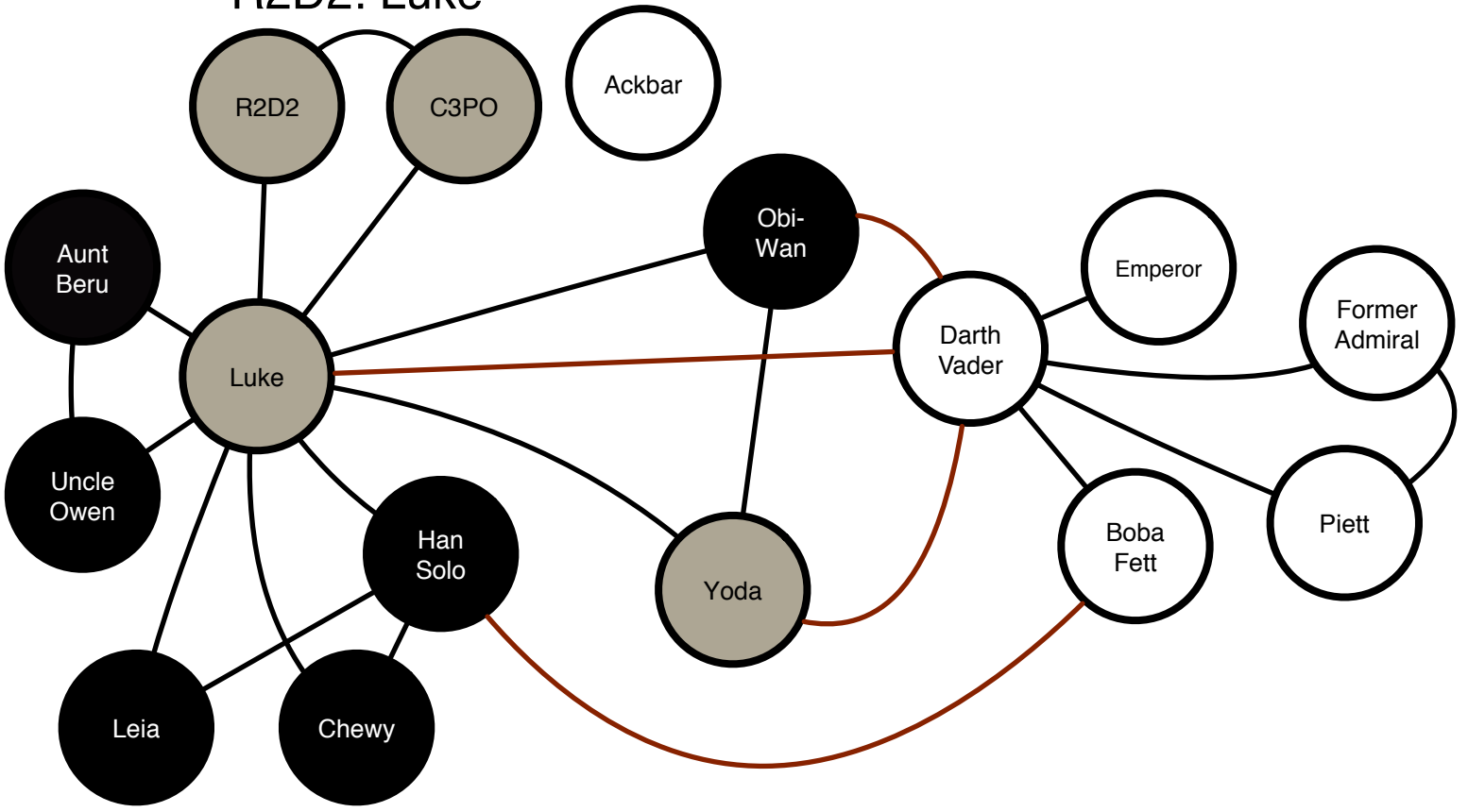
Yoda: Luke, Obi-Wan
Luke: Obi-Wan
C3PO: *empty*
R2D2: Luke



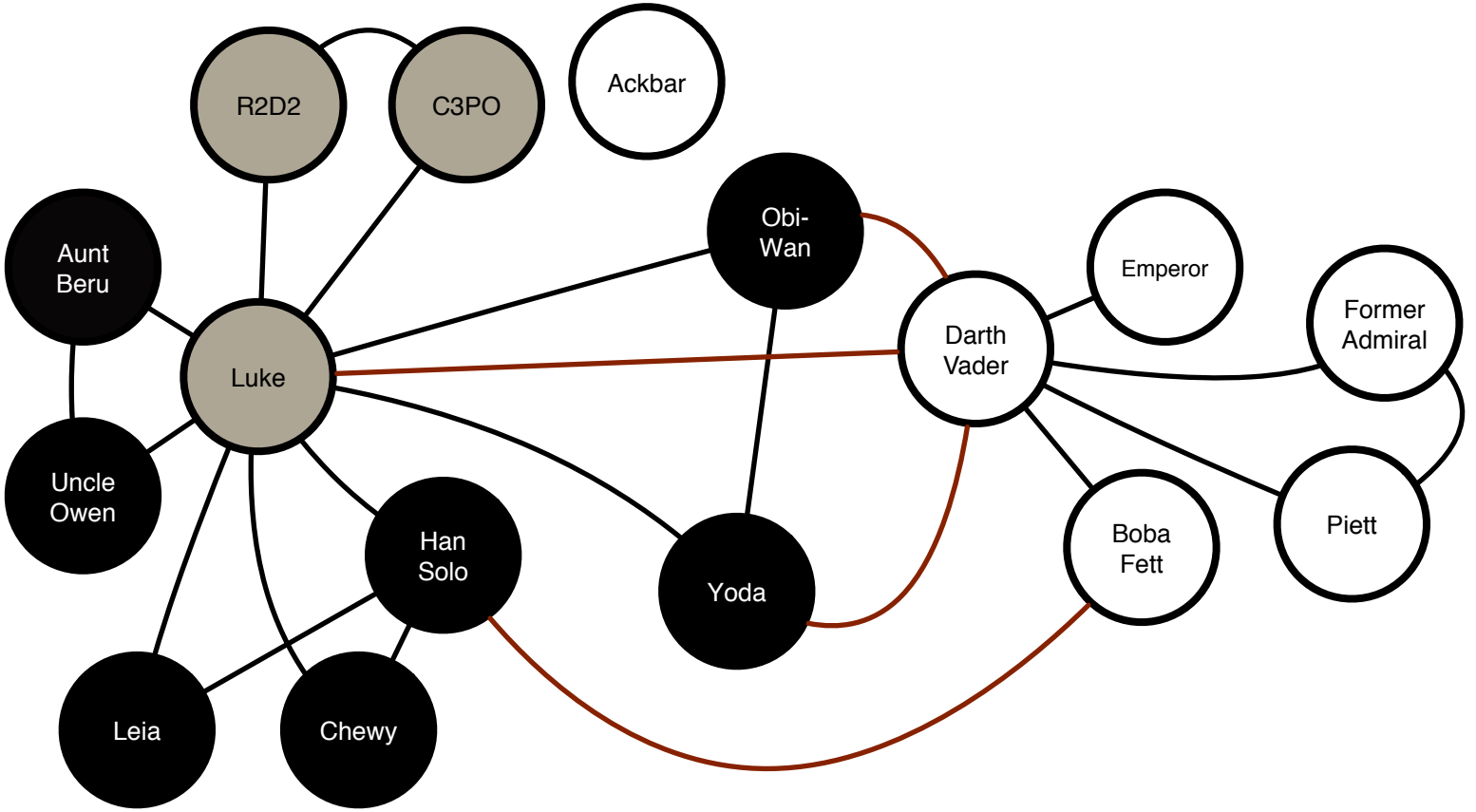
Obi-Wan: Luke, Yoda
Yoda: *empty*
Luke: Obi-Wan
C3PO: *empty*



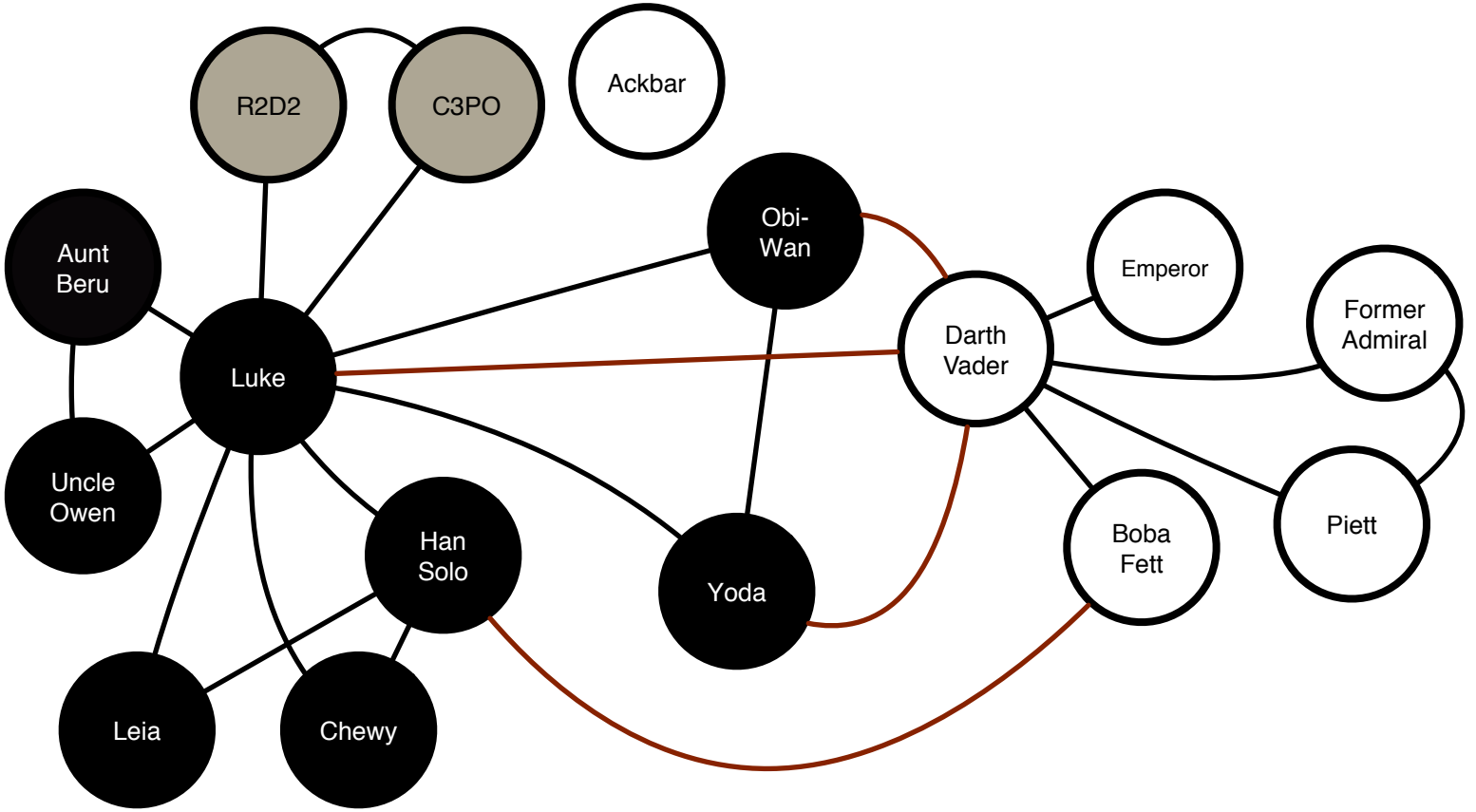
Yoda: *empty*
Luke: Obi-Wan
C3PO: *empty*
R2D2: Luke



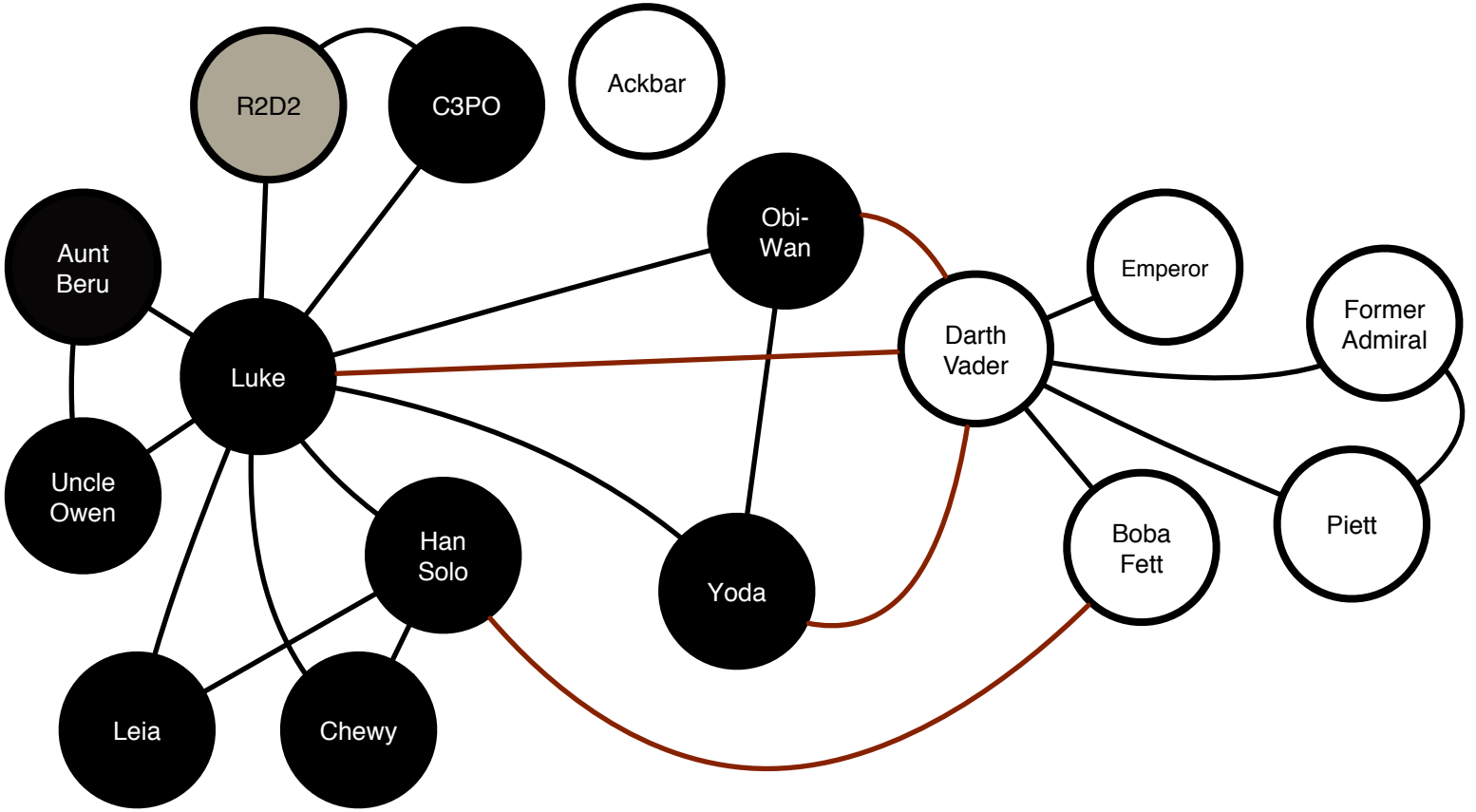
Luke: Obi-Wan
C3PO: *empty*
R2D2: Luke



C3PO: *empty*
R2D2: Luke



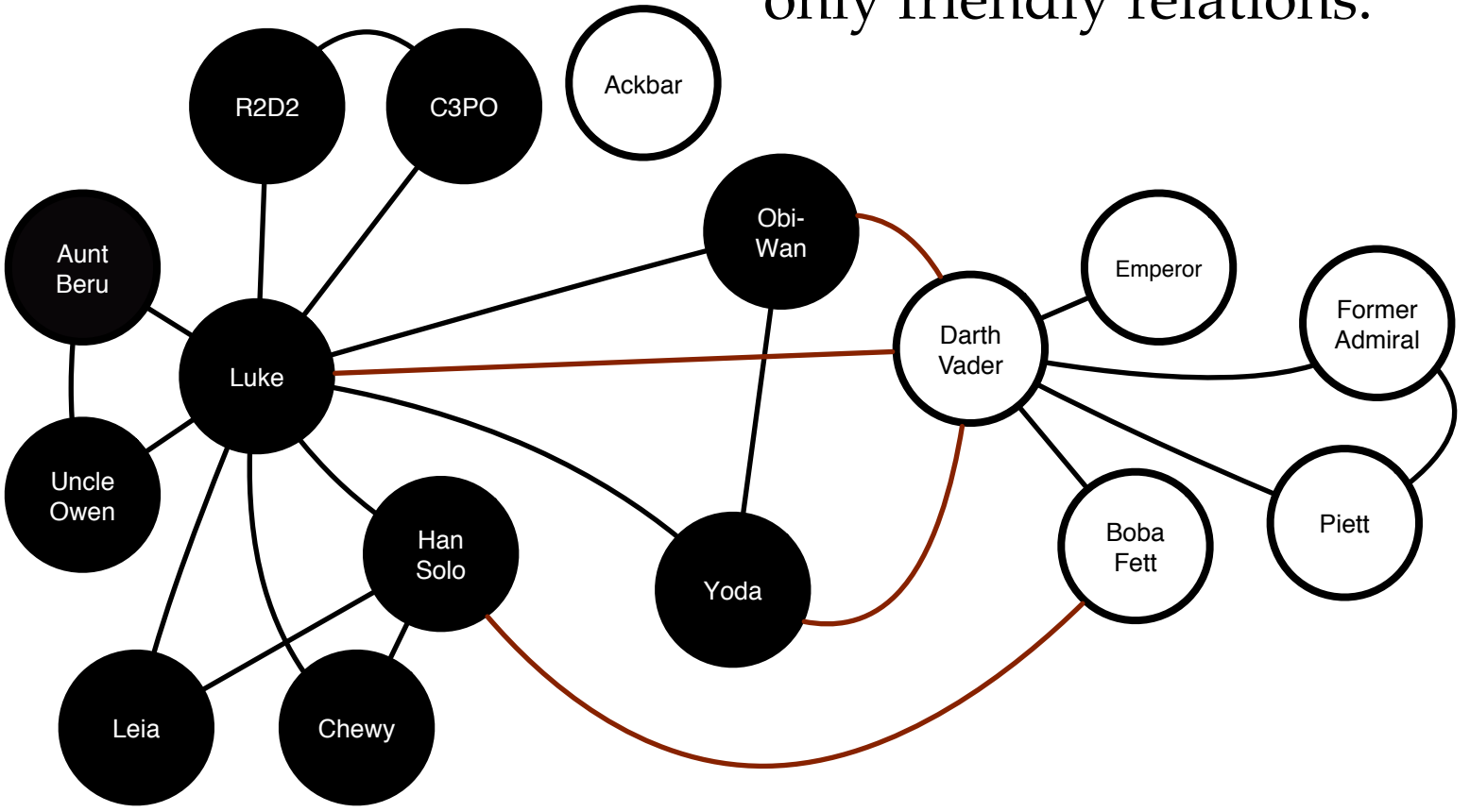
R2D2: Luke


R2D2
C3PO
Ackbar
Aunt Beru
Obi-Wan
Emperor
Former Admiral
Luke
Darth Vader
Uncle Owen
Han Solo
Yoda
Boba Fett
Piett
Leia
Chewy

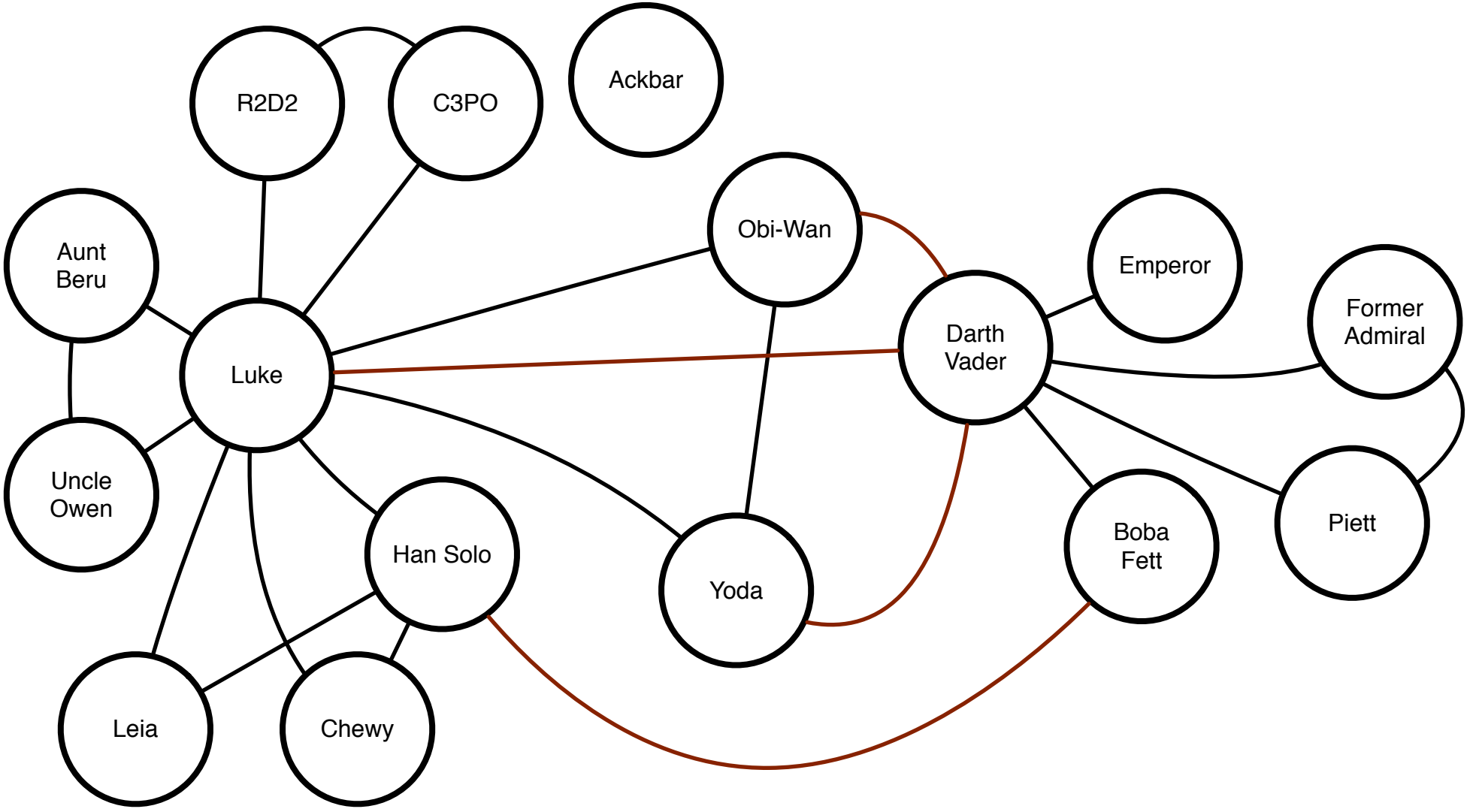
So now we have a complete Depth-First traversal of the graph using *edges marked as 'friendly'*.

R2D2: *empty*

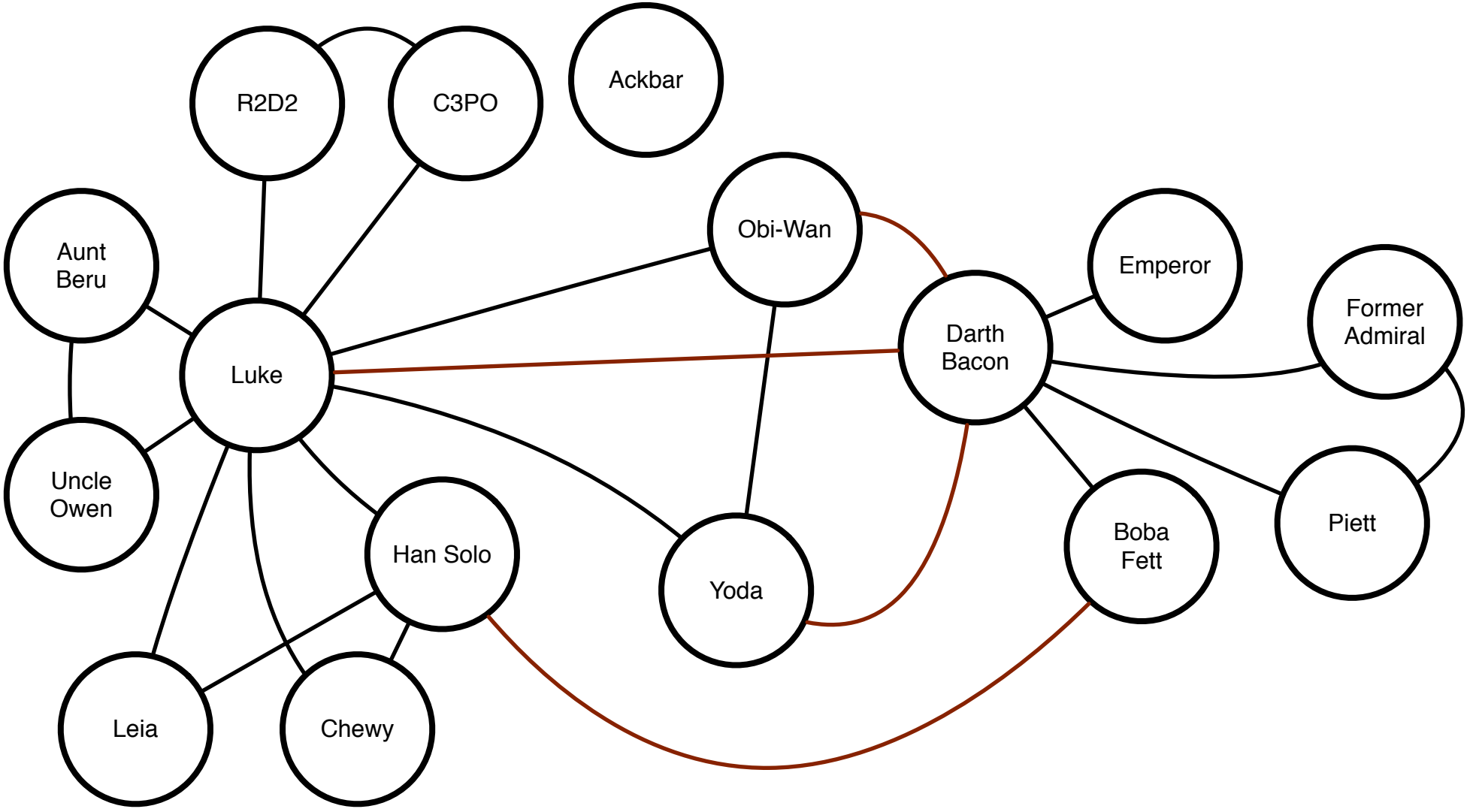
All the nodes that have been colored black are in the set of nodes that can be found starting from R2D2 following only friendly relations.



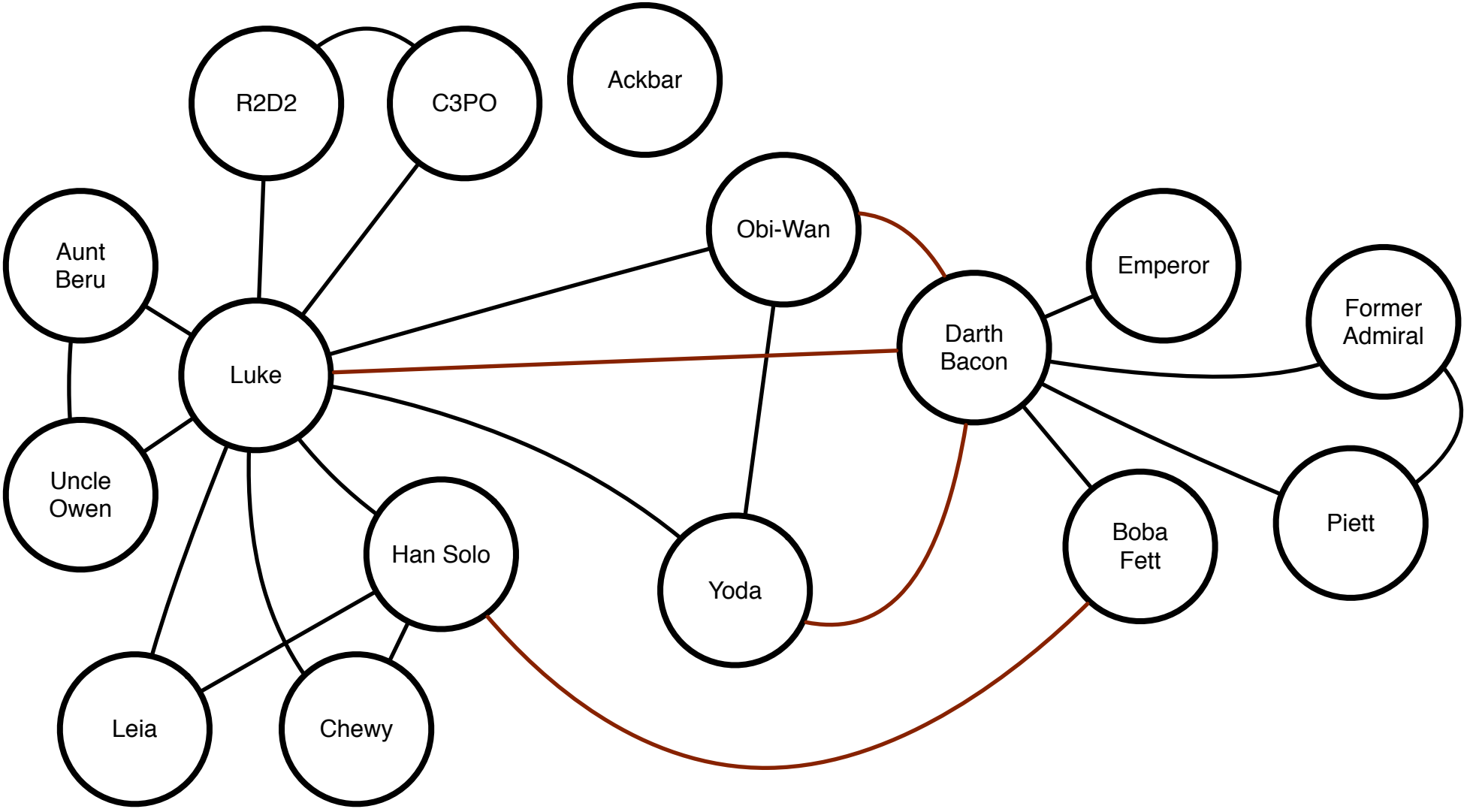
There's another basic form of graph algorithm: the breadth-first search. The idea here is to completely finish a node (color it black) before moving on to the children.



Task: We want to know how many levels of separation there are between Darth Vader and everybody else. For this task we will rename him *Darth Bacon* for reasons that should be obvious.



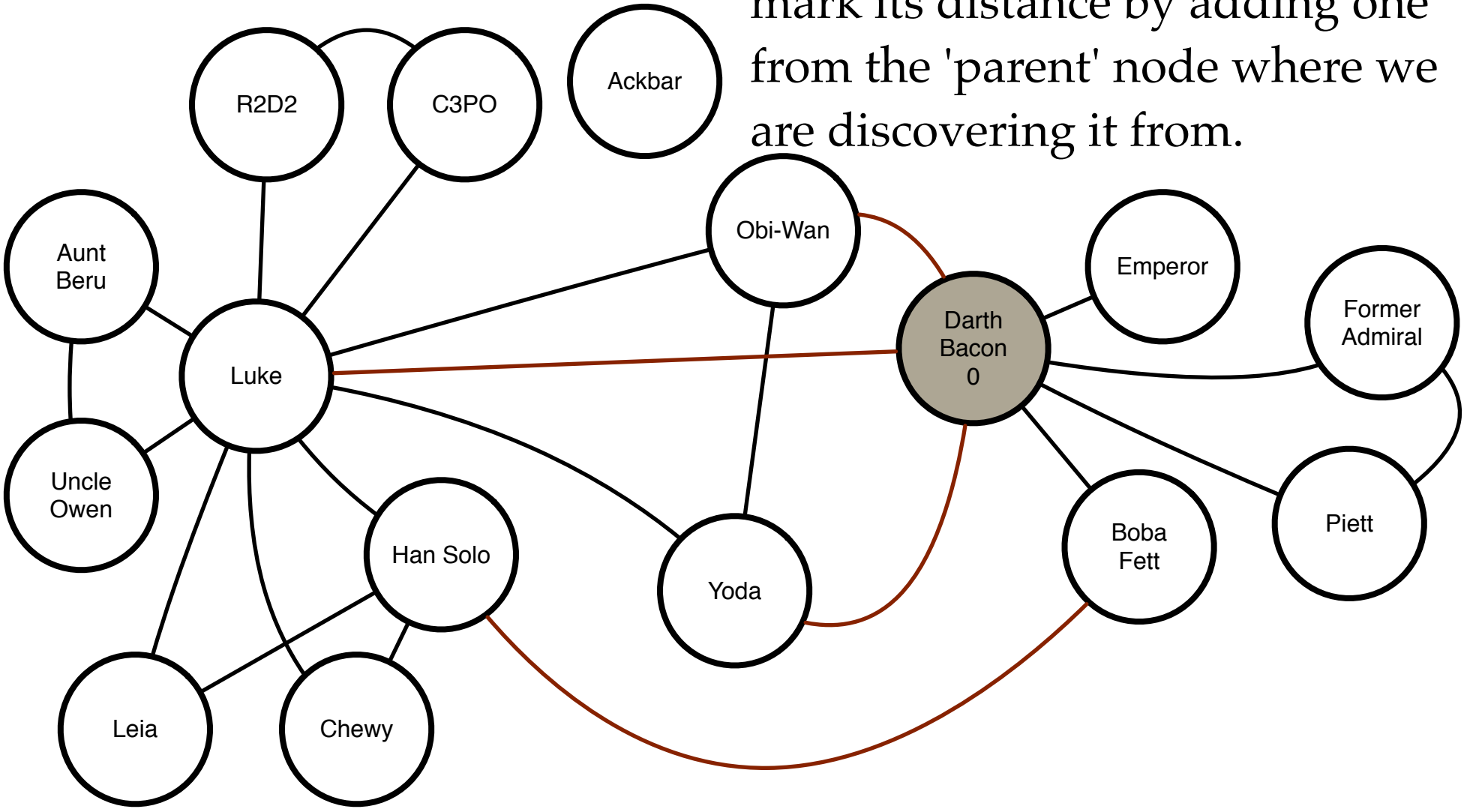
A Breadth-First search uses a queue that is shared among all node visitations. Remember: A queue is First In, First Out (FIFO).



Start at Darth Bacon by adding him to the Queue at distance=0.
When add a node to the queue, we 'mark' it (I'll color them).

Queue: Darth Bacon

When enqueuing a node, we mark its distance by adding one from the 'parent' node where we are discovering it from.



The basic algorithm is: take next item from the queue, visit it, add all of its unmarked children to the end of the queue. Continue until queue is empty.

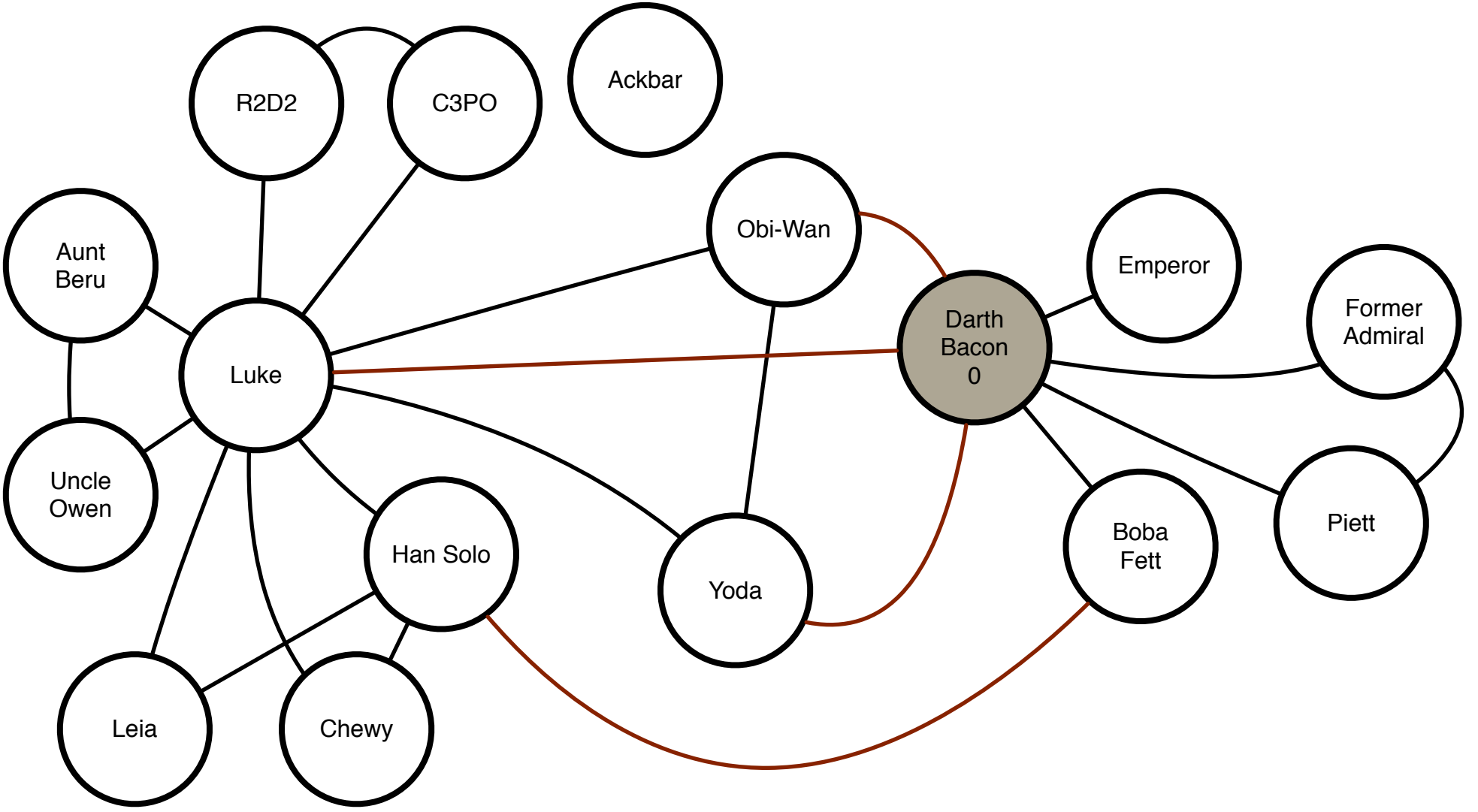
```
queue = empty queue
queue.add(start_node)
mark start_node gray
while (queue has stuff in it):
    node = queue.remove_first()
    visit node. this is your quality time with this particular node.
    for all edges e related to node:
        other_node = node on other end of e
        if (other_node is unmarked):
            queue.add(other_node)
            mark other_node
```

In our case, we will also maintain a *depth* value. When we add an unmarked node to the queue, we also set its depth value to one plus the current node's depth value. This is how we will count the minimum distance from the start node to all the other nodes.

So: remove 1st item in queue.

Current Item: Darth Bacon

Queue: *empty*

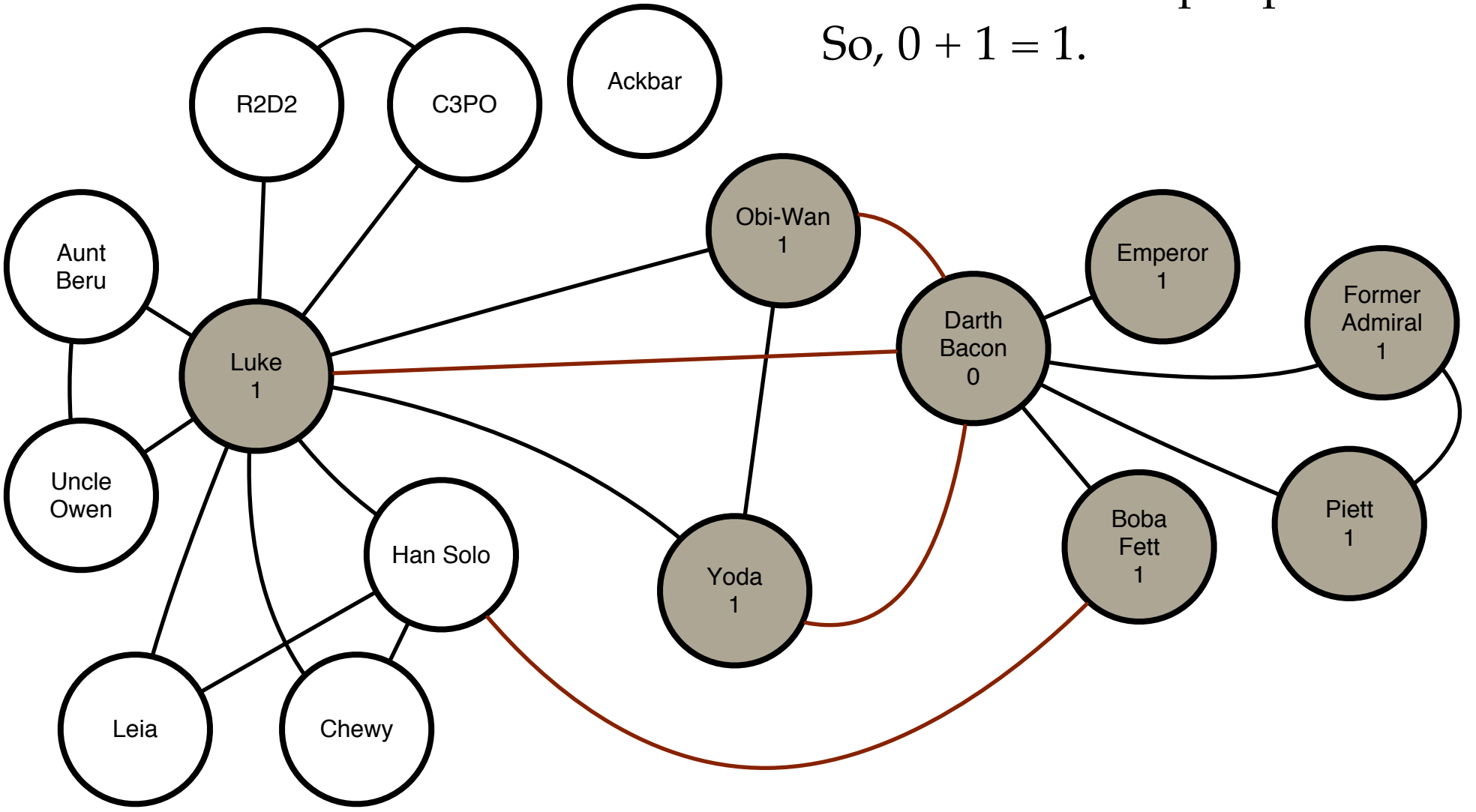


Then: enqueue unmarked neighbors (ignore edge color here).

Current Item: Darth Bacon

Queue: Obi-Wan, Emperor, Former Admiral, Piett, Boba Fett, Yoda, Luke

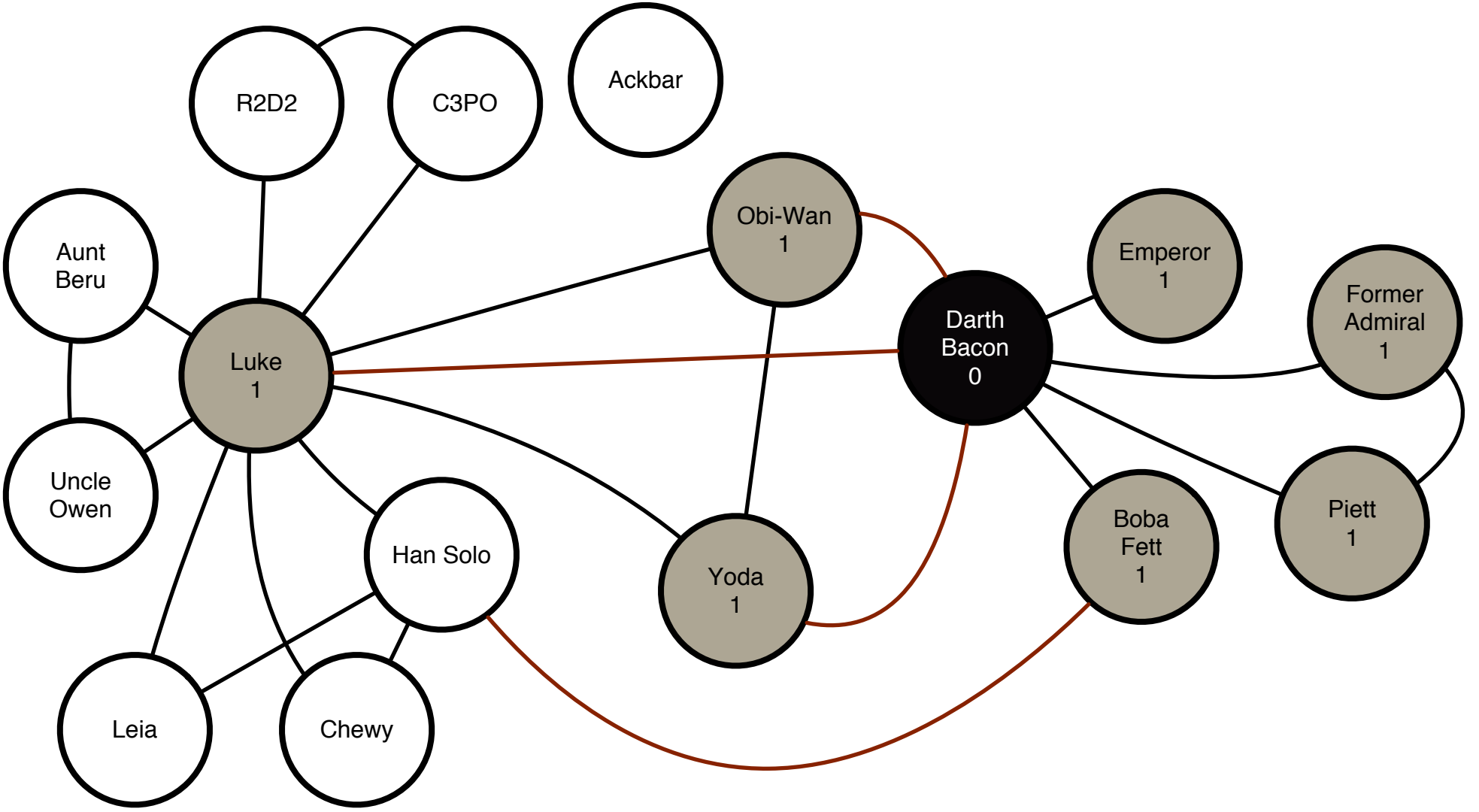
Enqueued nodes get the current item's depth plus one. So, 0 + 1 = 1.



After: mark current item as done. This is optional.

Current Item: Darth Bacon

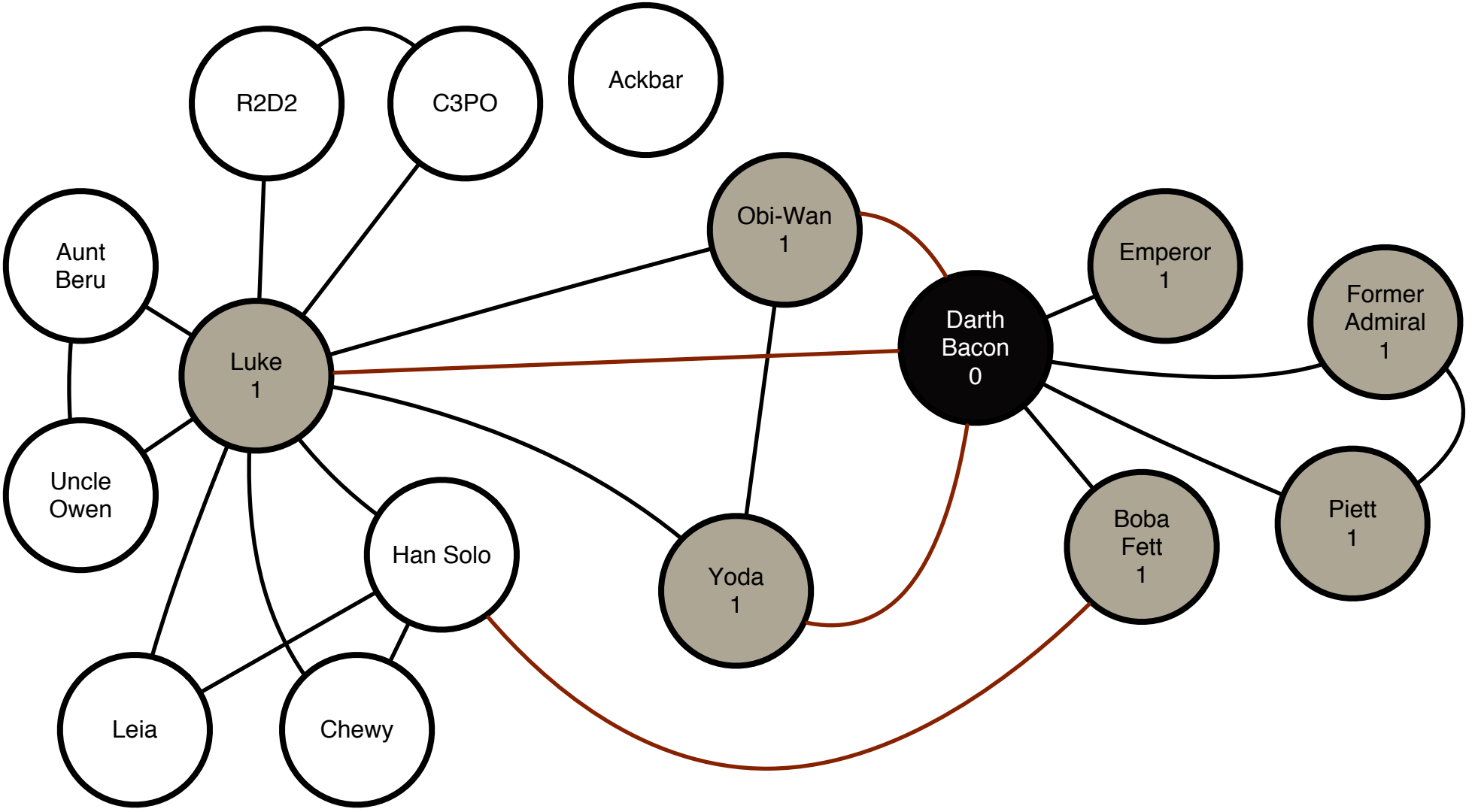
Queue: Obi-Wan, Emperor, Former Admiral, Piett, Boba Fett, Yoda, Luke



Continue. Dequeue 1st item (Obi-Wan). No unmarked neighbors.

Current Item: Obi-Wan

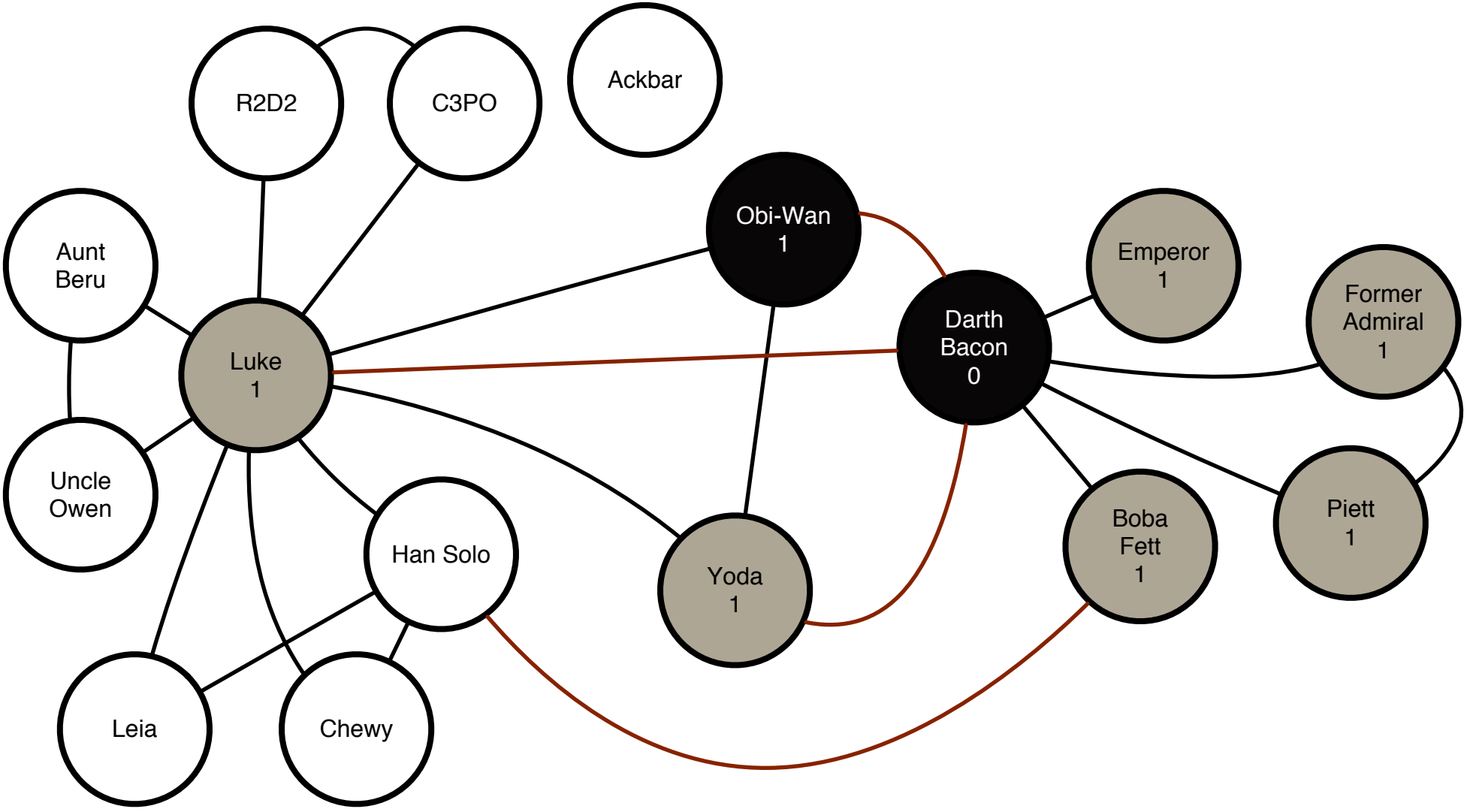
Queue: Emperor, Former Admiral, Piett, Boba Fett, Yoda, Luke



Continue. Dequeue next (Emperor). No unmarked neighbors.

Current Item: Emperor

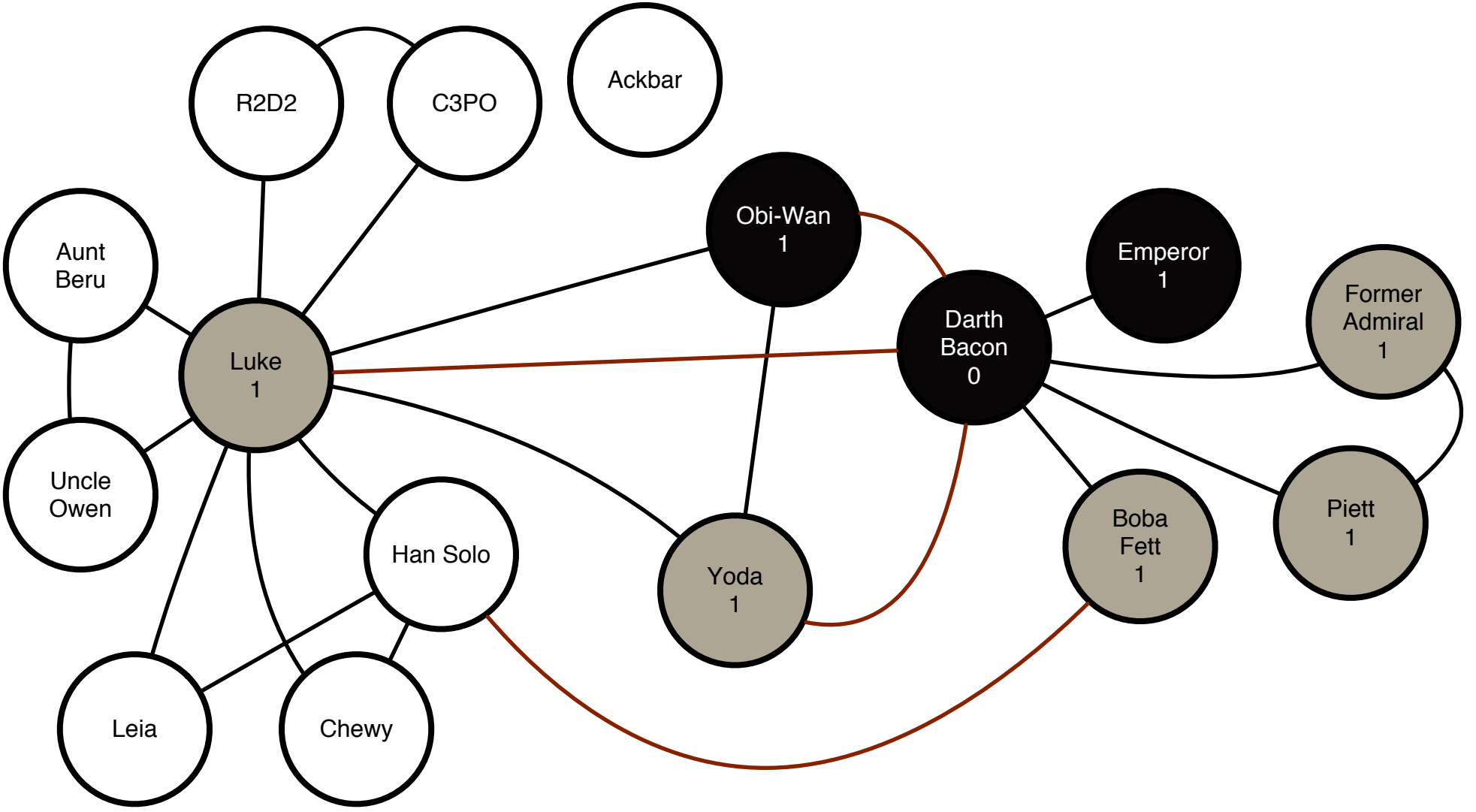
Queue: Former Admiral, Piett, Boba Fett, Yoda, Luke



Continue. Dequeue next (Admiral). No unmarked neighbors.

Current Item: Former Admiral

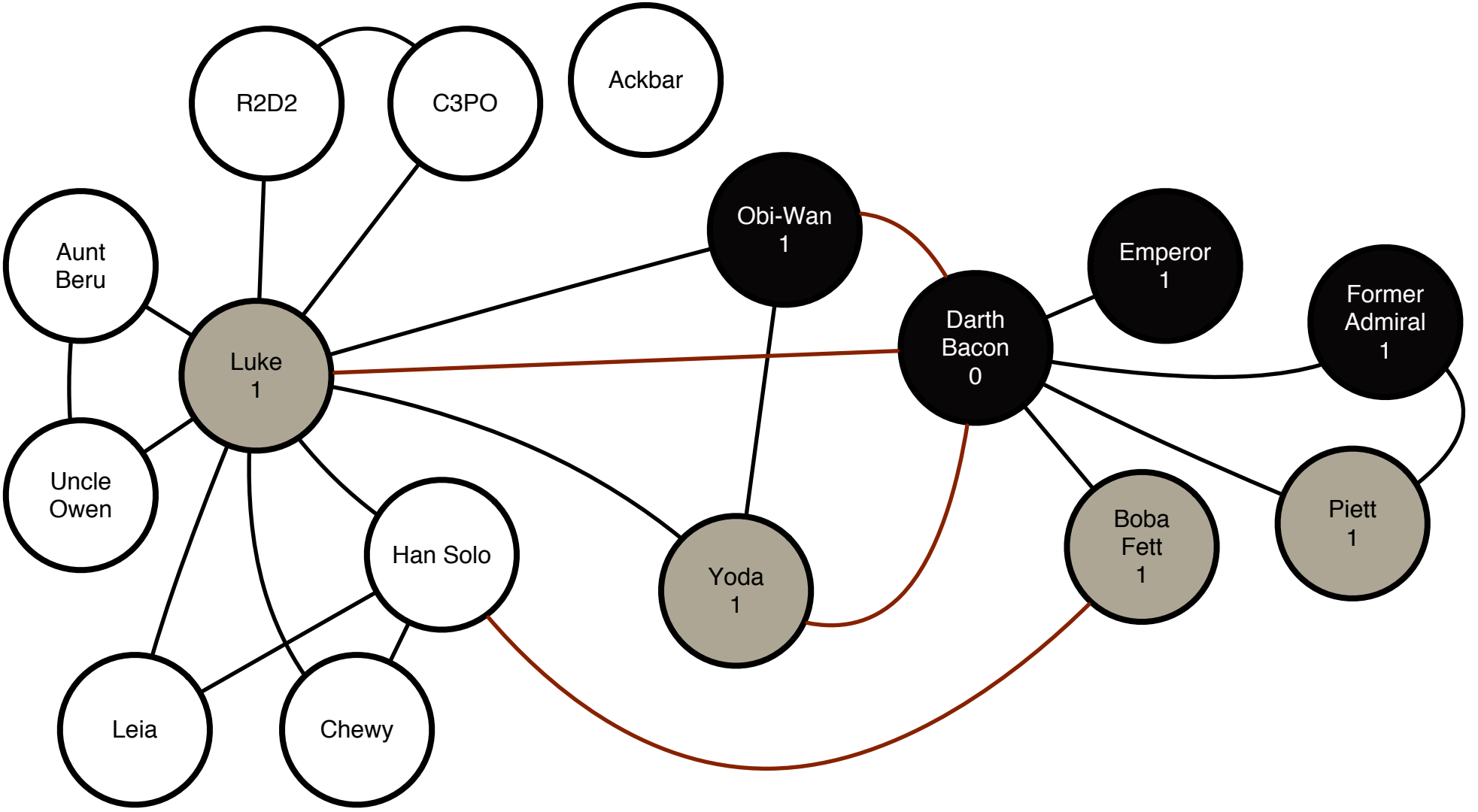
Queue: Piett, Boba Fett, Yoda, Luke



Continue. Dequeue next (Piett). No unmarked neighbors.

Current Item: Piett

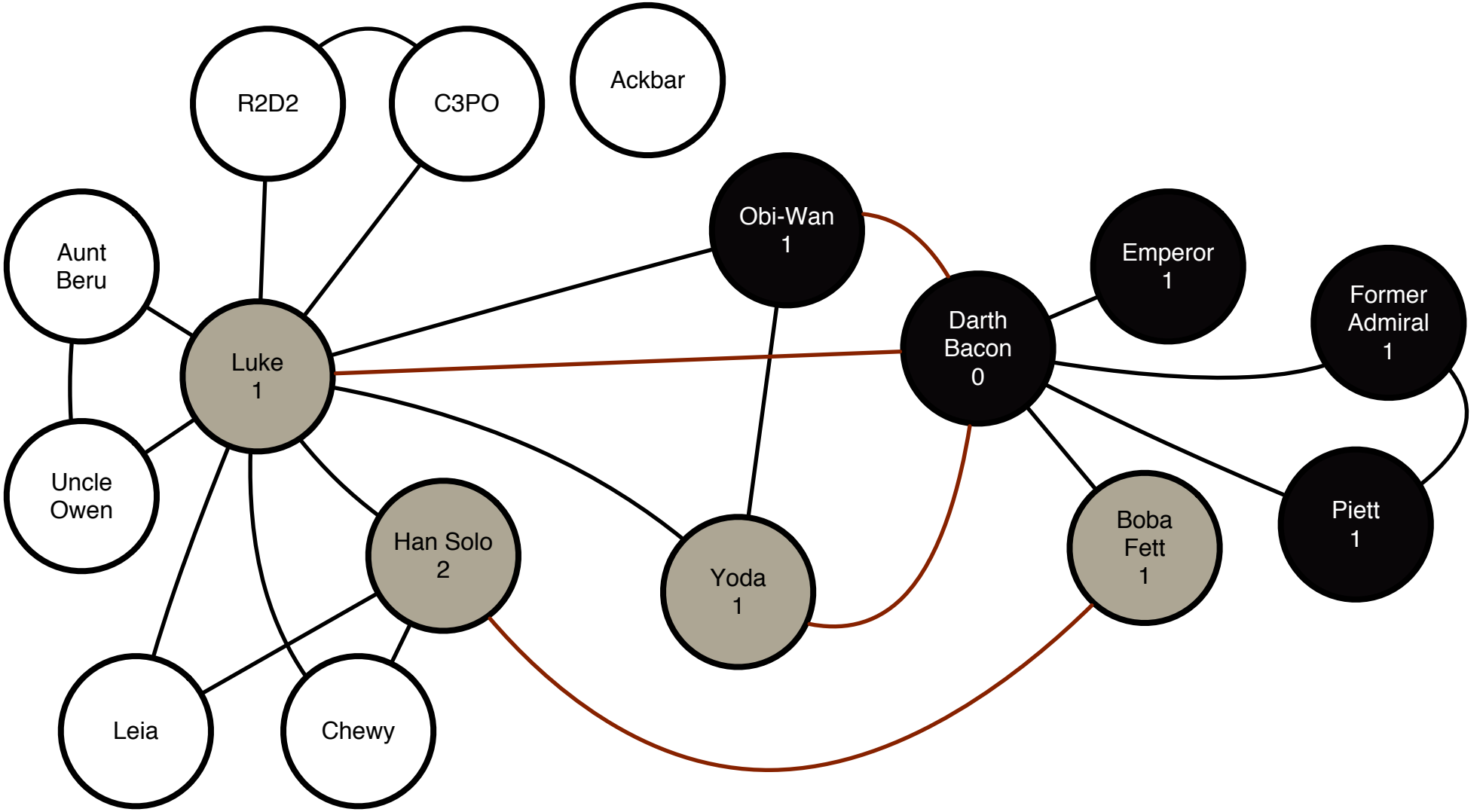
Queue: Boba Fett, Yoda, Luke



Continue. Dequeue next (Boba Fett). Neighboring Han Solo!

Current Item: Boba Fett

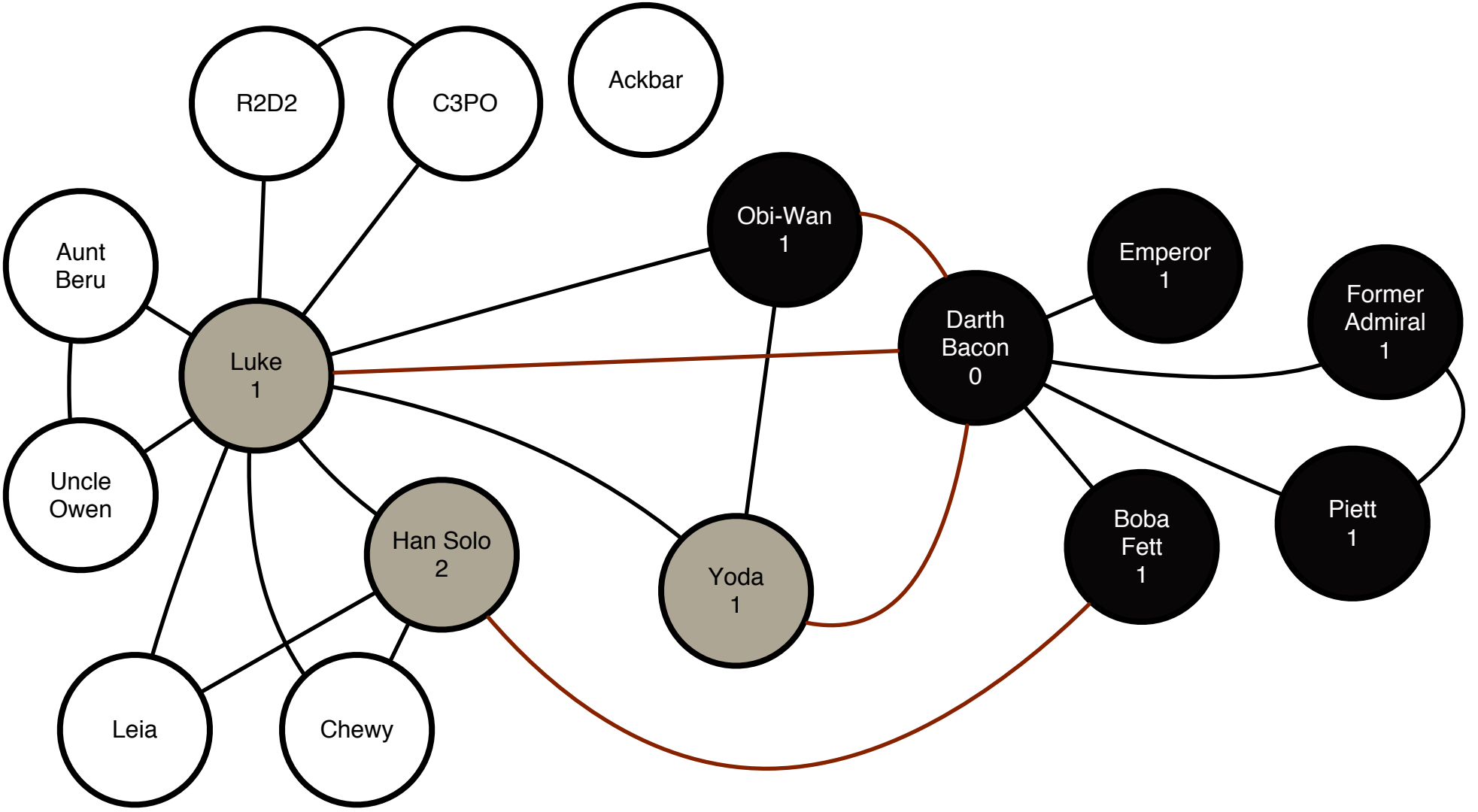
Queue: Yoda, Luke, Han Solo



Continue. Dequeue next (Yoda). No unmarked neighbors.

Current Item: Yoda

Queue: Luke, Han Solo

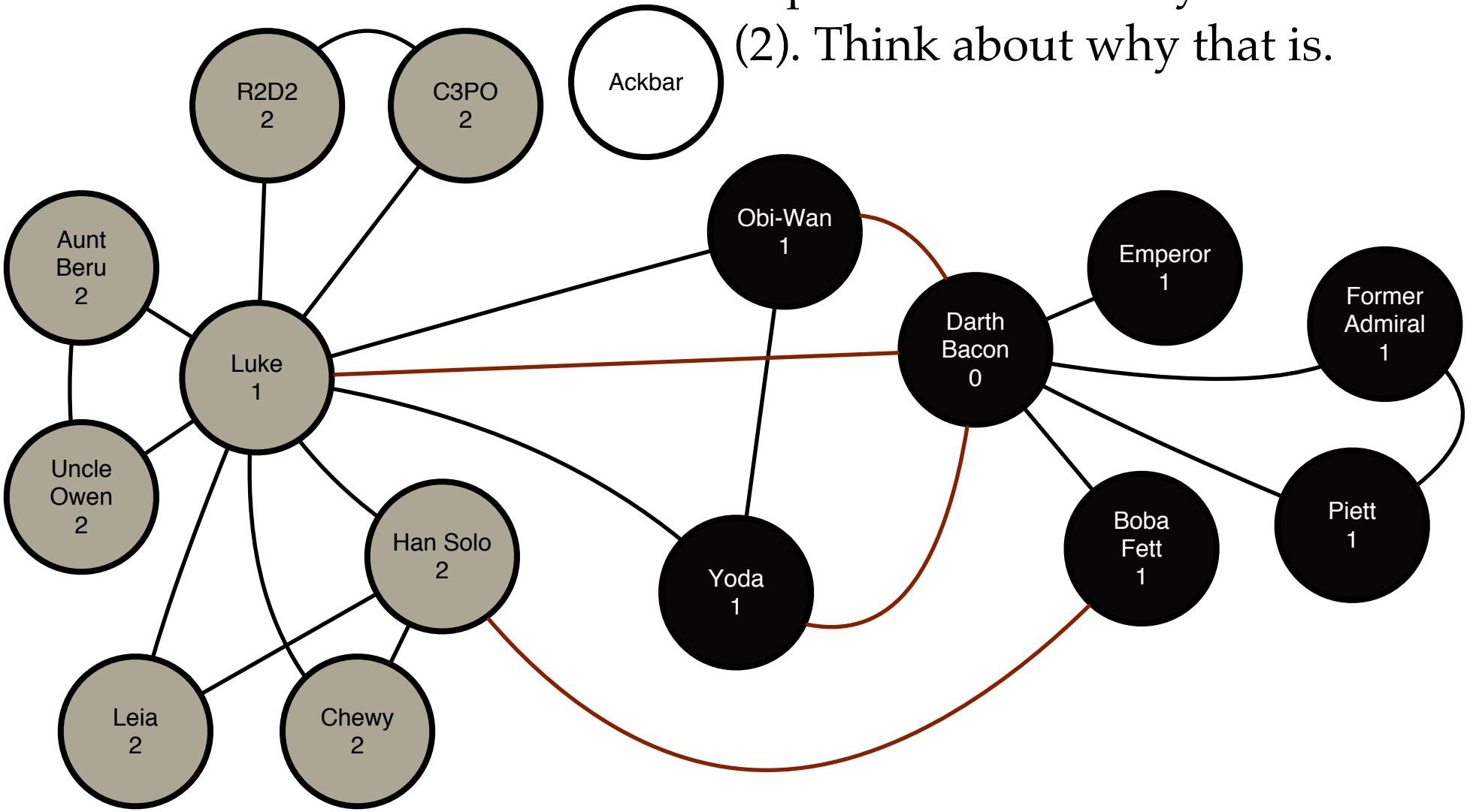


Continue. Dequeue next (Luke). Many unmarked neighbors.

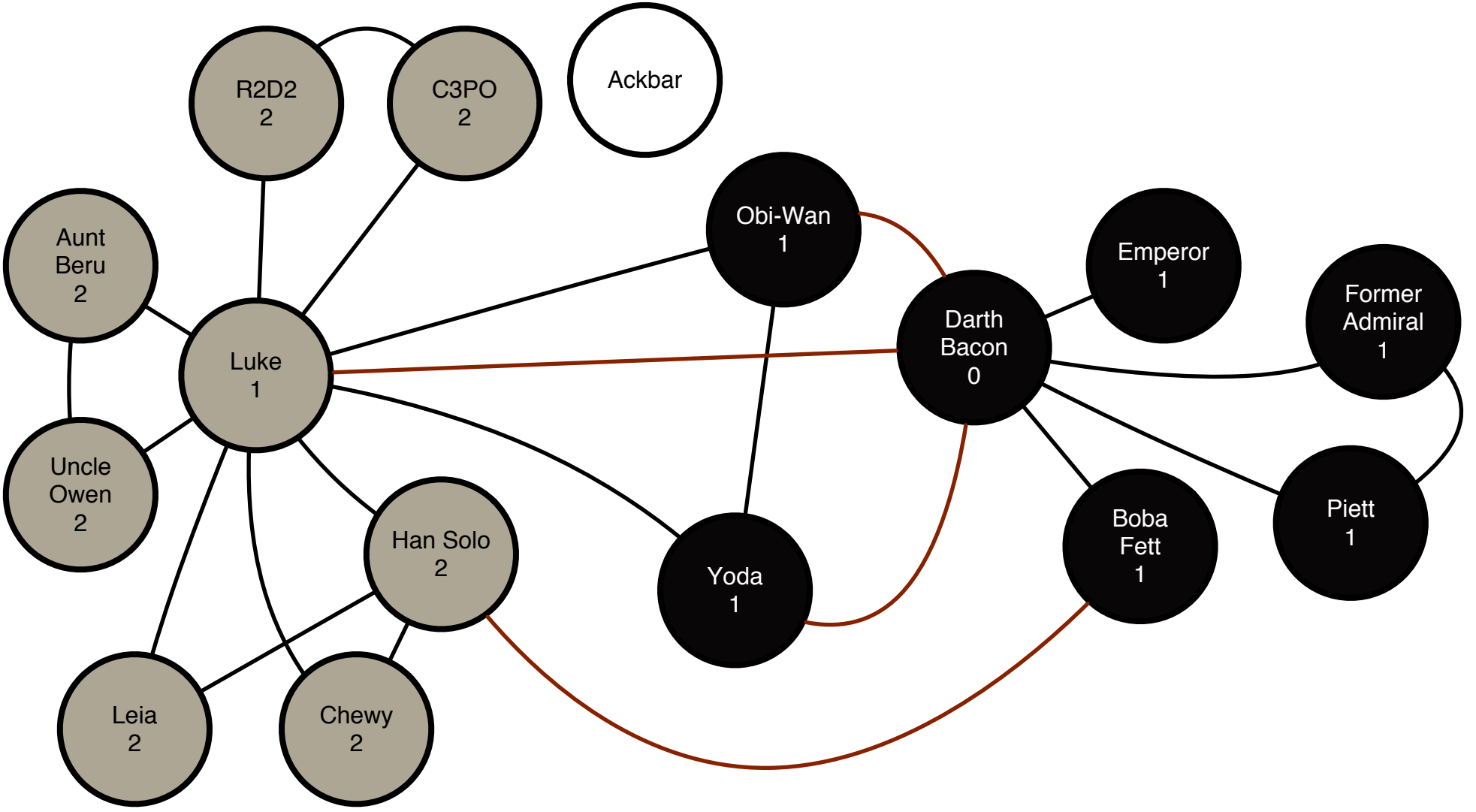
Current Item: Luke

Queue: Han Solo, Chewy, Leia, Uncle Owen, Aunt Beru, R2D2, C3PO

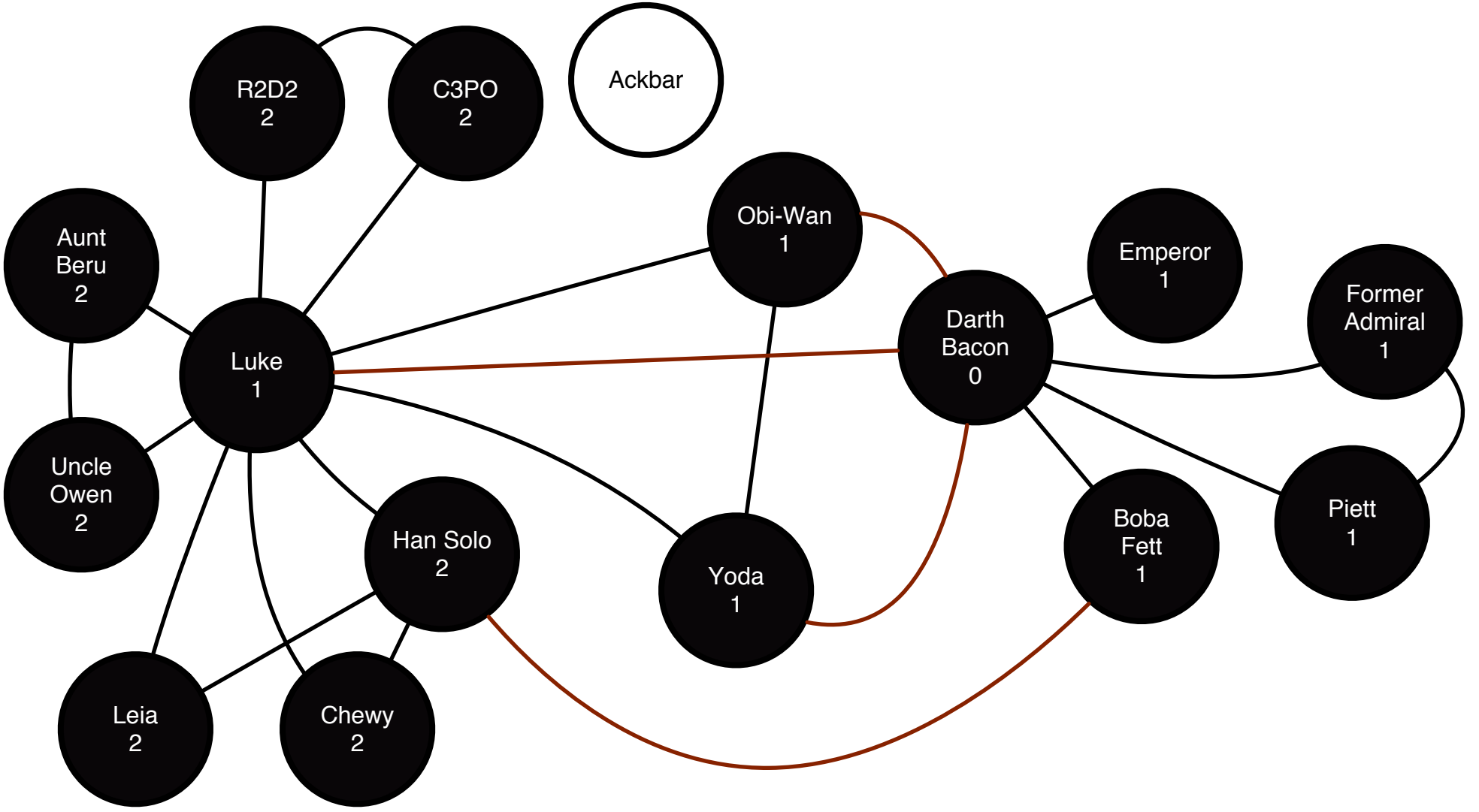
Notice how Han Solo has the same depth as all the newly added nodes (2). Think about why that is.



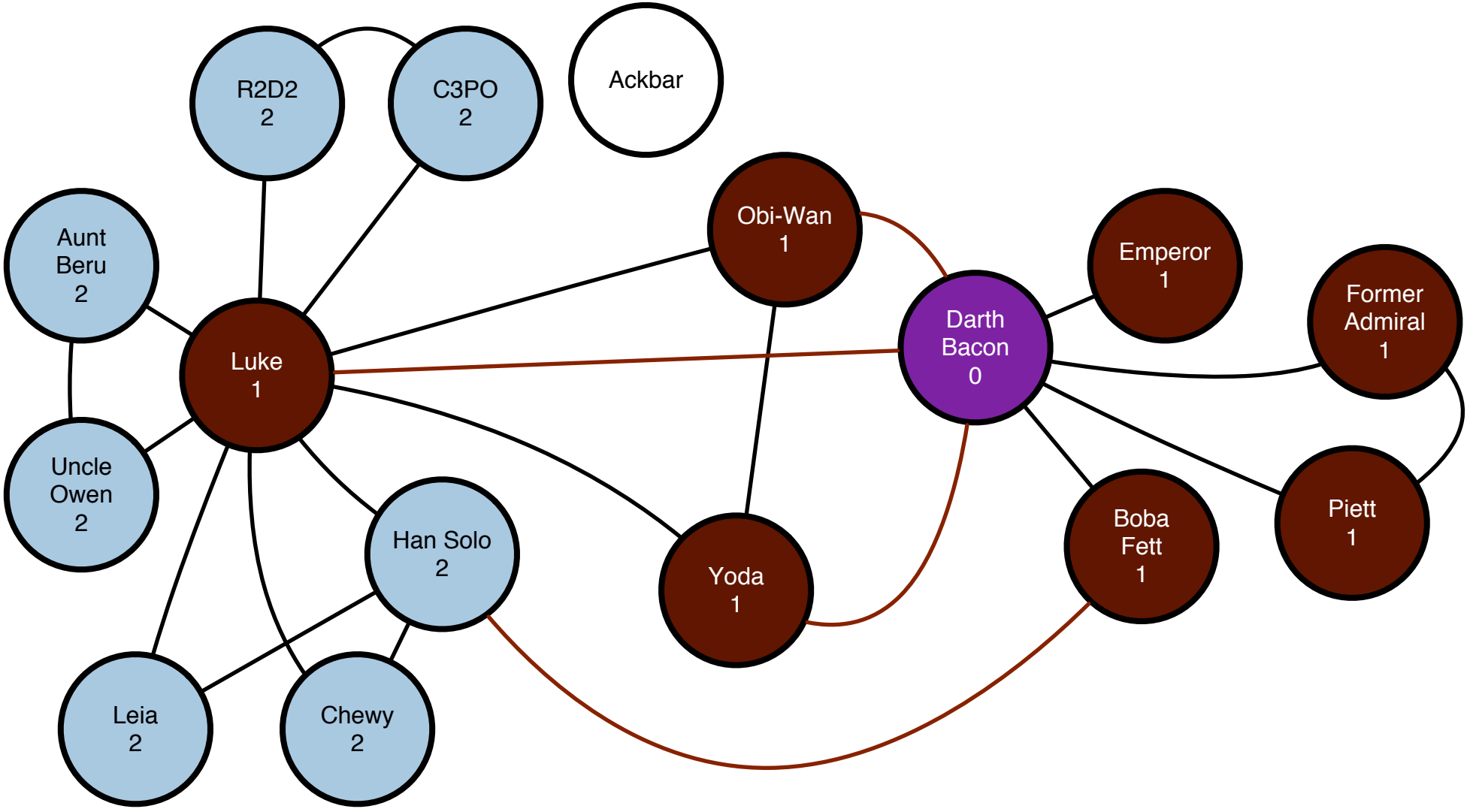
From here, we won't discover any more unmarked edges.



Eventually we will have visited all nodes. Except for poor Ackbar. He's aware of our trap, and will stay out of it.



I can re-color the nodes according to how far many levels of separation there are to Darth Bacon.



Almost everything you will do with a graph is some variation on the Depth-First and Breadth-First search algorithms.

I write this is very large font so you know I'm serious.