breadth-first search 6

In this chapter

- You learn how to model a network using a new, abstract data structure: graphs.
- You learn breadth-first search, an algorithm you can run on graphs to answer questions like, "What's the shortest path to go to X?"
- You learn about directed versus undirected graphs.
- You learn topological sort, a different kind of sorting algorithm that exposes dependencies between nodes.

This chapter introduces graphs. First, I'll talk about what graphs are (they don't involve an X or Y axis). Then I'll show you your first graph algorithm. It's called *breadth-first search* (BFS).

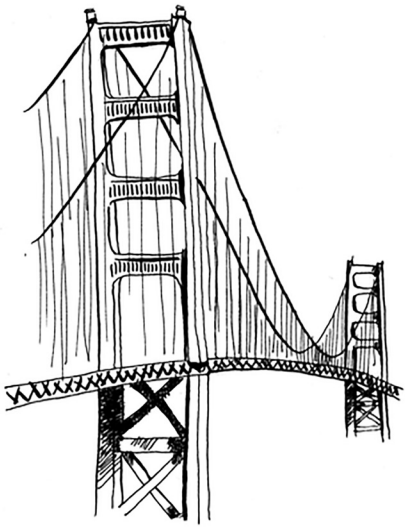
Breadth-first search allows you to find the shortest distance between two things. But shortest distance can mean a lot of things! You can use breadth-first search to

- Write a checkers AI that calculates the fewest moves to victory

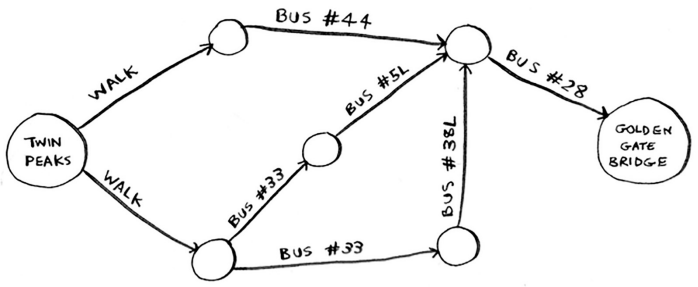
- Write a spell checker (fewest edits from your misspelling to a real word—for example, READED -> READER is one edit)
- Find the doctor closest to you in your network

Graph algorithms are some of the most useful algorithms I know. Make sure you read the next few chapters carefully—these are algorithms you'll be able to apply again and again.

Introduction to graphs

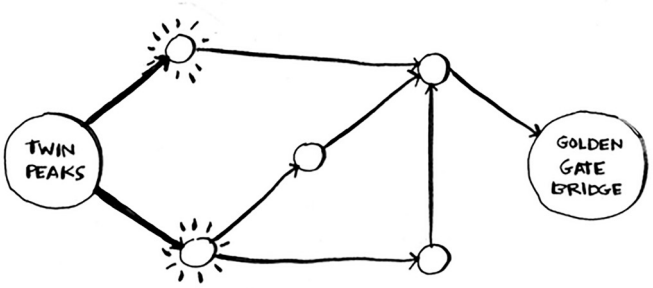


Suppose you're in San Francisco, and you want to go from Twin Peaks to the Golden Gate Bridge. You want to get there by bus, with the minimum number of transfers. Here are your options.

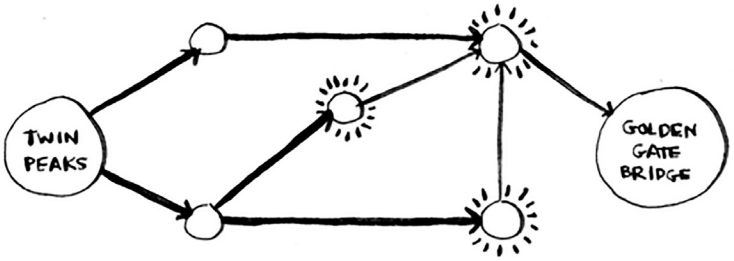


What's your algorithm to find the path with the fewest steps?

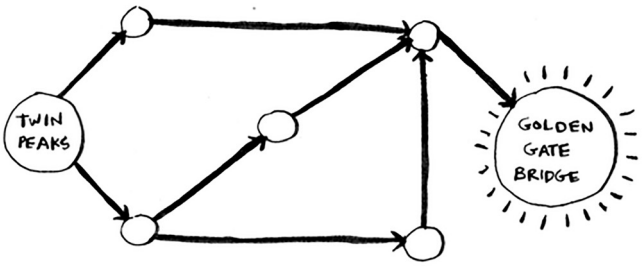
Well, can you get there in one step? Here are all the places you can get to in one step.



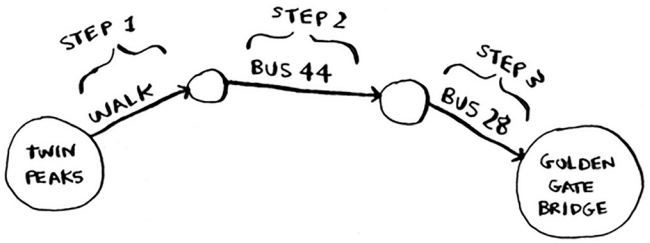
The bridge isn't highlighted; you can't get there in one step. Can you get there in two steps?



Again, the bridge isn't there, so you can't get to the bridge in two steps. What about three steps?



Aha! Now the Golden Gate Bridge shows up. So it takes three steps to get from Twin Peaks to the bridge using this route.



There are other routes that will get you to the bridge too, but they're longer (four steps). The algorithm found that the shortest route to the bridge is three steps long. This type of problem is called a *shortest-path problem*. You're always trying to find the shortest something. It could be the shortest route to your friend's house. It could be the smallest number of moves to checkmate in a game of chess. The algorithm to solve a shortest-path problem is called *breadth-first search*.

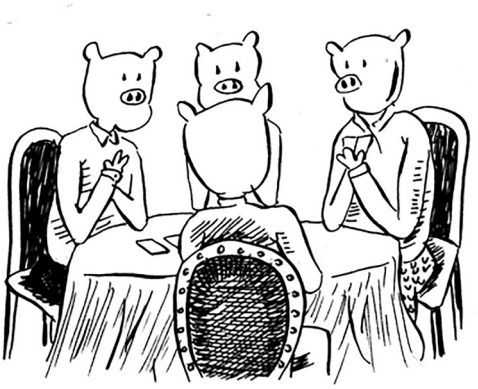
To figure out how to get from Twin Peaks to the Golden Gate Bridge, there are two steps:

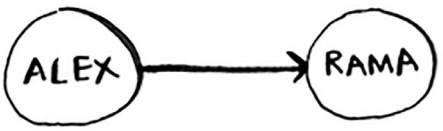
1. Model the problem as a graph.
2. Solve the problem using breadth-first search.

Next I'll cover what graphs are. Then I'll go into breadth-first search in more detail.

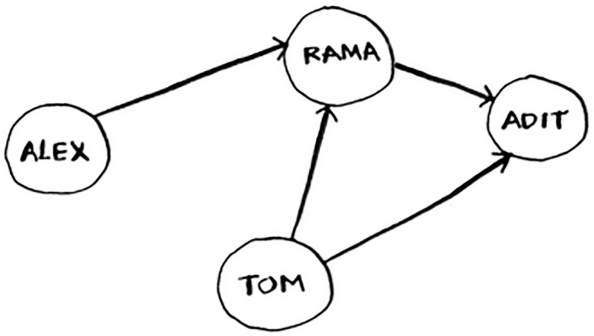
What is a graph?

A graph models a set of connections. For example, suppose you and your friends are playing poker, and you want to model who owes whom money. Here's how you could say, "Alex owes Rama money."



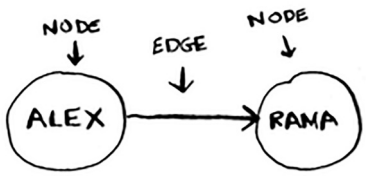


The full graph could look something like this.



Graph of people who owe other people poker money

Alex owes Rama money, Tom owes Adit money, and so on. Each graph is made up of *nodes* and *edges*.



That's all there is to it! Graphs are made up of nodes and edges. A node can be directly connected to many other nodes. Those nodes are called its *neighbors*. In this graph, Rama is Alex's neighbor. Adit isn't Alex's neighbor, because they aren't directly connected. But Adit is Rama's and Tom's neighbor.

Graphs are a way to model how different things are connected to one another. Now let's see breadth-first search in action.

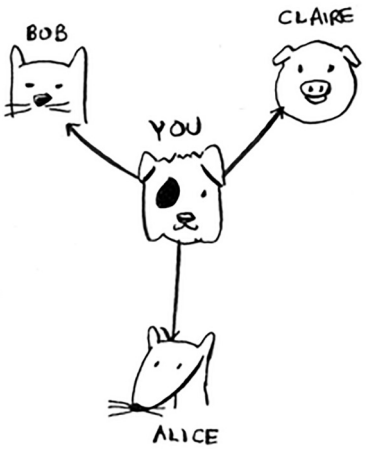
Breadth-first search

We looked at a search algorithm in chapter 1: binary search. Breadth-first search is a different kind of search algorithm: one that runs on graphs. It can help answer two types of questions:

- Question type 1: Is there a path from node A to node B?
- Question type 2: What is the shortest path from node A to node B?

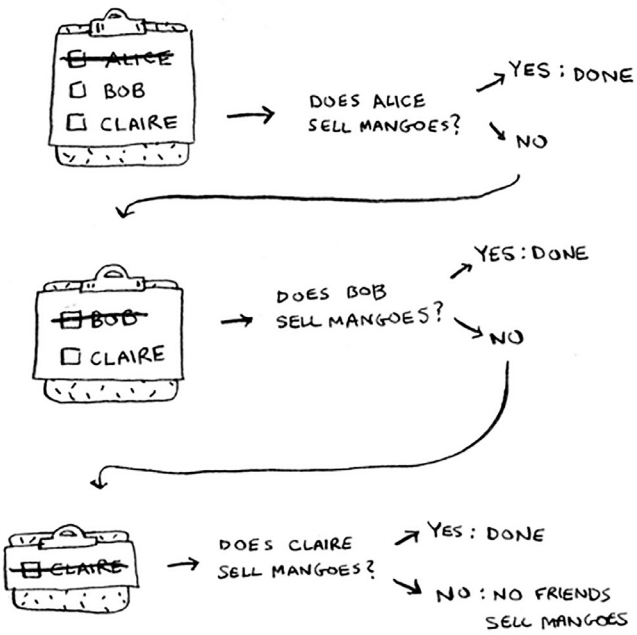
You already saw breadth-first search once, when you calculated the shortest route from Twin Peaks to the Golden Gate Bridge. That was a question of type 2: "What is the shortest path?" Now let's look at the algorithm in more detail. You'll ask a question of type 1: "Is there a path?"

Suppose you're the proud owner of a mango farm. You're looking for a mango seller who can sell your mangoes. Are you connected to a mango seller on Facebook? Well, you can search through your friends.

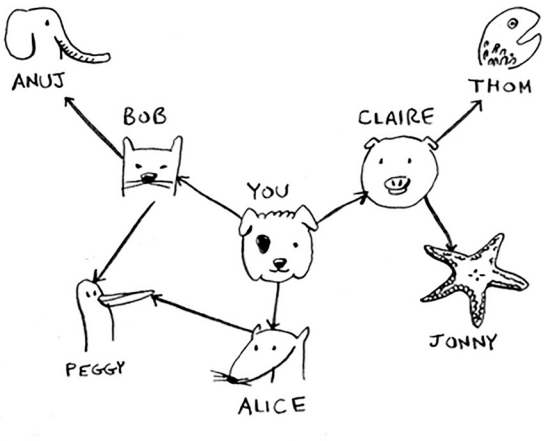


This search is pretty straightforward. First, make a list of friends to search.

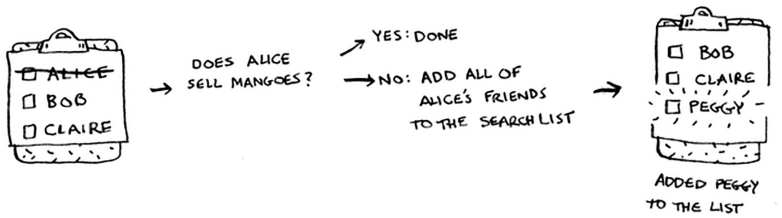
Now, go to each person in the list and check whether that person sells mangoes.



Suppose none of your friends are mango sellers. Now you have to search through your friends' friends.



Each time you search for someone from the list, add all of their friends to the list.



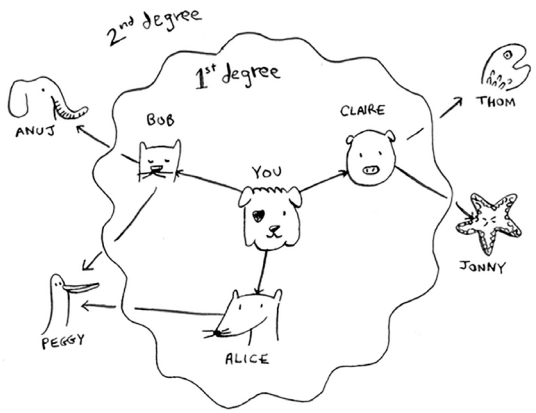
This way, you not only search your friends, but you search their friends, too. Remember, the goal is to find one mango seller in your network. So if Alice isn't a mango seller, you add her friends to the list, too. That means you'll eventually search her friends—and then their friends, and so on. With this algorithm, you'll search your entire network until you come across a mango seller. This algorithm is breadth-first search.

Finding the shortest path

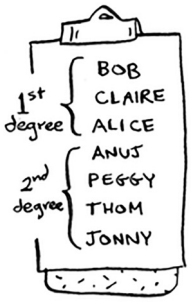
As a recap, these are the two questions that breadth-first search can answer for you:

- Question type 1: Is there a path from node A to node B? (Is there a mango seller in your network?)
- Question type 2: What is the shortest path from node A to node B? (Who is the closest mango seller?)

You saw how to answer question 1; now let's try to answer question 2. Can you find the closest mango seller? For example, your friends are first-degree connections, and their friends are second-degree connections.



You'd prefer a first-degree connection to a second-degree connection, and a second-degree connection to a third-degree connection, and so on. So you shouldn't search any second-degree connections before you make sure you don't have a first-degree connection who is a mango seller. Well, breadth-first search already does this! The way breadth-first search works, the search radiates out from the starting point. So you'll check first-degree connections before second-degree connections. Pop quiz: who will be checked first, Claire or Anuj? Answer: Claire is a first-degree connection, and Anuj is a second-degree connection. So Claire will be checked before Anuj.



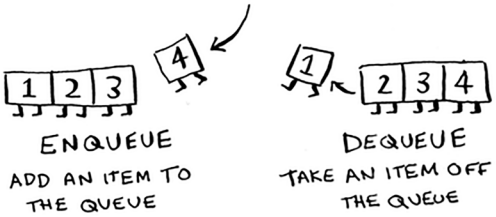
Another way to see this is, first-degree connections are added to the search list before second-degree connections.

You just go down the list and check people to see whether each one is a mango seller. The first-degree connections will be searched before the second-degree connections, so you'll find the mango seller closest to you. Breadth-first search not only finds a path from A to B, it also finds the shortest path.

Notice that this only works if you search people in the same order in which they're added. That is, if Claire was added to the list before Anuj, Claire needs to be searched before Anuj. What happens if you search Anuj before Claire, and they're both mango sellers? Well, Anuj is a second-degree contact, and Claire is a first-degree contact. You end up with a mango seller who isn't the closest to you in your network. So you need to search people in the order that they're added. There's a data structure for this: it's called *a queue*.

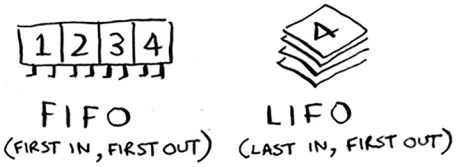
Queues

A queue works exactly like it does in real life. Suppose you and your friend are queueing up at the bus stop. If you're before him in the queue, you get on the bus first. A queue works the same way. Queues are similar to stacks. You can't access random elements in the queue. Instead, there are two only operations, *enqueue* and *dequeue*.



If you enqueue two items to the list, the first item you added will be dequeued before the second item. You can use this for your search list! People who are added to the list first will be dequeued and searched first.

The queue is called a *FIFO* data structure: First In, First Out. In contrast, a stack is a *LIFO* data structure: Last In, First Out.

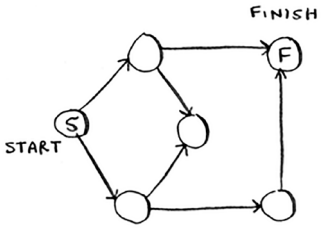


Now that you know how a queue works, let's implement breadth-first search!

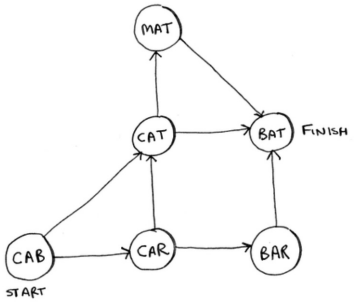
EXERCISES

Run the breadth-first search algorithm on each of these graphs to find the solution.

6.1 Find the length of the shortest path from start to finish.

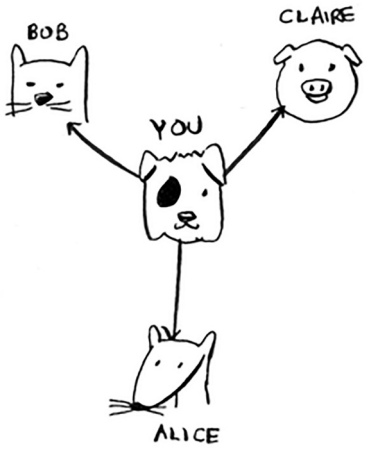


6.2 Find the length of the shortest path from "cab" to "bat".



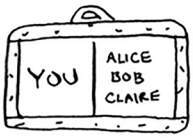
Implementing the graph

First, you need to implement the graph in code. A graph consists of several nodes.



And each node is connected to neighboring nodes. How do you express a relationship like "you -> bob"? Luckily, you know a data structure that lets you express relationships: *a hash table*!

Remember, a hash table allows you to map a key to a value. In this case, you want to map a node to all of its neighbors.

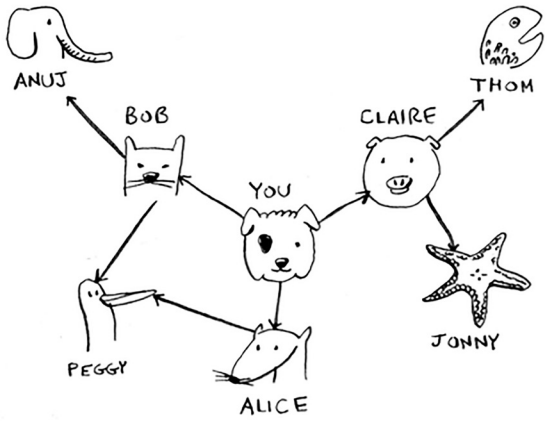


Here's how you'd write it in Python:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

Notice that "you" is mapped to an array. So `graph["you"]` will give you an array of all the neighbors of "you".

A graph is just a bunch of nodes and edges, so this is all you need to have a graph in Python. What about a bigger graph, like this one?



Here it is as Python code:

```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
graph["bob"] = ["anuj", "peggy"]
graph["alice"] = ["peggy"]
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
graph["peggy"] = []
graph["thom"] = []
graph["jonny"] = []
```

Pop quiz: does it matter what order you add the key/value pairs in? Does it matter if you write

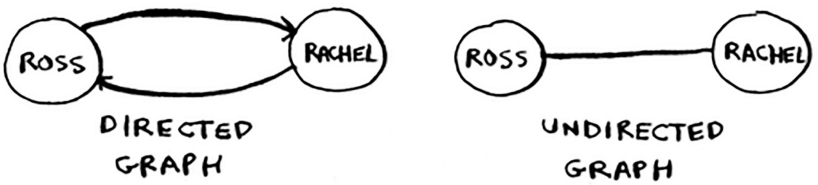
```
graph["claire"] = ["thom", "jonny"]
graph["anuj"] = []
```

instead of

```
graph["anuj"] = []
graph["claire"] = ["thom", "jonny"]
```

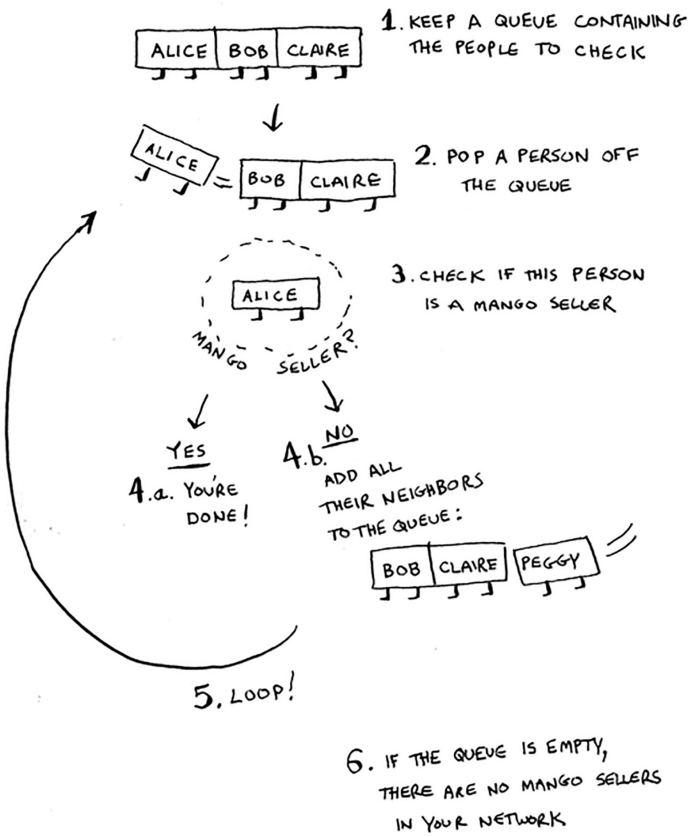
Think back to the previous chapter. Answer: It doesn't matter. Hash tables have no ordering, so it doesn't matter what order you add key/value pairs in.

Anuj, Peggy, Thom, and Jonny don't have any neighbors. They have arrows pointing to them, but no arrows from them to someone else. This is called a *directed graph*—the relationship is only one way. So Anuj is Bob's neighbor, but Bob isn't Anuj's neighbor. An undirected graph doesn't have any arrows, and both nodes are each other's neighbors. For example, both of these graphs are equal.



Implementing the algorithm

To recap, here's how the implementation will work.



> **Note**
>
> When updating queues, I use the terms *enqueue* and *dequeue*. You'll also encounter the terms *push* and *pop*. *Push* is almost always the same thing as *enqueue*, and *pop* is almost always the same thing as *dequeue*.

Make a queue to start. In Python, you use the double-ended queue (`deque`) function for this:

```
from collections import deque
search_queue = deque()          <-- Creates a new queue
search_queue += graph["you"]    <-- Adds all of your neighbors to the search queue
```

Remember, `graph["you"]` will give you a list of all your neighbors, like `["alice", "bob", "claire"]`. Those all get added to the search queue.

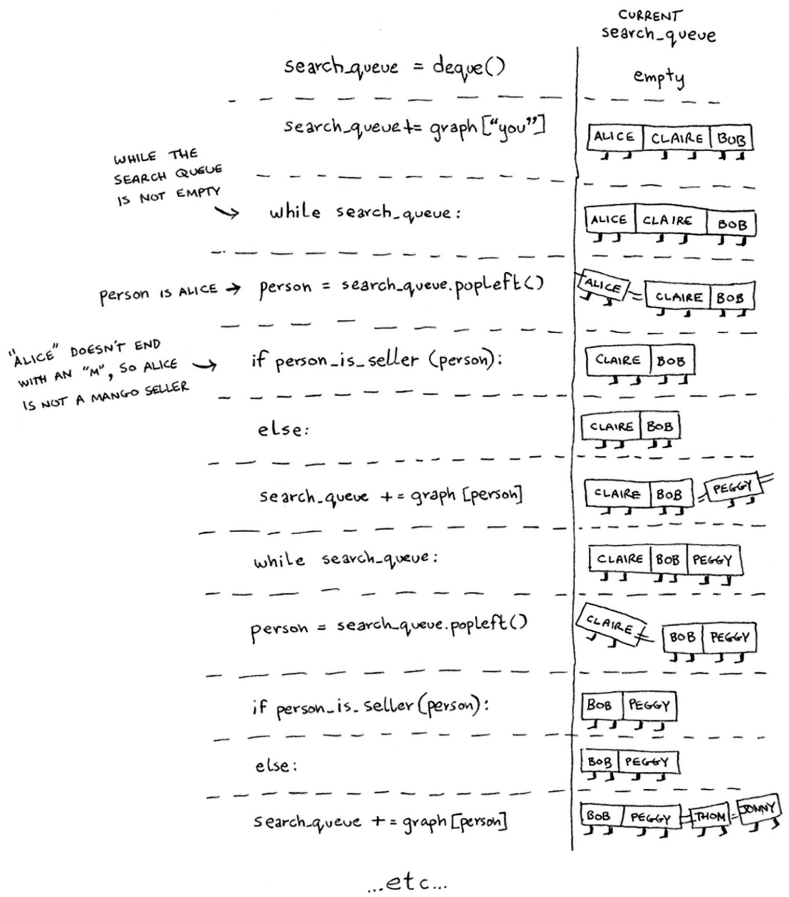
Let's see the rest:

```
while search_queue:                          ◄ While the queue isn't empty ...
    person = search_queue.popleft()          ◄ ... grabs the first person off the queue
    if person_is_seller(person):             ◄ Checks whether the person is a mango seller
        print person + " is a mango seller!" ◄ Yes, they're a mango seller.
        return True
    else:
        search_queue += graph[person]        ◄ No, they aren't. Add all of this
return False                                   person's friends to the search queue.
                                             ◄ If you reached here, no one in
                                               the queue was a mango seller.
```

One final thing: you still need a `person_is_seller` function to tell you when someone is a mango seller. Here's one:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

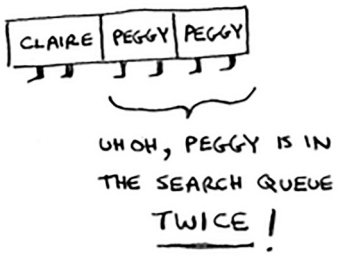
This function checks whether the person's name ends with the letter *m*. If it does, they're a mango seller. Kind of a silly way to do it, but it'll do for this example. Now let's see the breadth-first search in action.



And so on. The algorithm will keep going until either

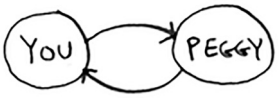
- A mango seller is found, or
- The queue becomes empty, in which case there is no mango seller.

Alice and Bob share a friend: Peggy. So Peggy will be added to the queue twice: once when you add Alice's friends, and again when you add Bob's friends. You'll end up with two Peggys in the search queue.



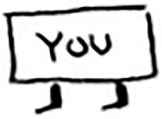
But you only need to check Peggy once to see whether she's a mango seller. If you check her twice, you're doing unnecessary, extra work. So once you search a person, you should mark that person as searched and not search them again.

If you don't do this, you could also end up in an infinite loop. Suppose the mango seller graph looked like this.



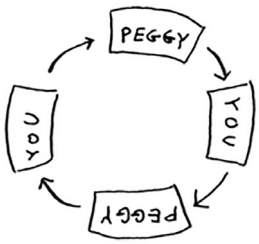
To start, the search queue contains all of your neighbors.

Now you check Peggy. She isn't a mango seller, so you add all of her neighbors to the search queue.



Next, you check yourself. You're not a mango seller, so you add all of your neighbors to the search queue.

And so on. This will be an infinite loop, because the search queue will keep going from you to Peggy.



Before checking a person, it's important to make sure they haven't been checked already. To do that, you'll keep a list of people you've already checked.

Here's the final code for breadth-first search, taking that into account:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:
            if person_is_seller(person):
                print person + " is a mango seller!"
                return True
            else:
                search_queue += graph[person]
                searched.append(person)
    return False

search("you")
```

This array is how you keep track of which people you've searched before.

Only search this person if you haven't already searched them.

Marks this person as searched

Try running this code yourself. Maybe try changing the `person_is_seller` function to something more meaningful, and see if it prints what you expect.

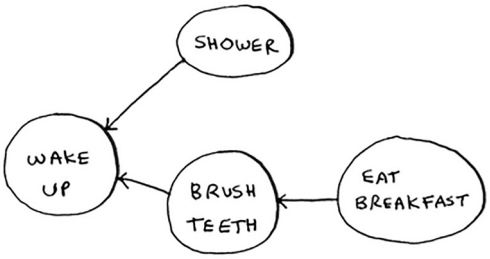
Running time

If you search your entire network for a mango seller, that means you'll follow each edge (remember, an edge is the arrow or connection from one person to another). So the running time is at least O(number of edges).

You also keep a queue of every person to search. Adding one person to the queue takes constant time: O(1). Doing this for every person will take O(number of people) total. Breadth-first search takes O(number of people + number of edges), and it's more commonly written as O(V+E) (V for number of vertices, E for number of edges).

EXERCISE

Here's a small graph of my morning routine.



It tells you that I can't eat breakfast until I've brushed my teeth. So "eat breakfast" *depends on* "brush teeth".

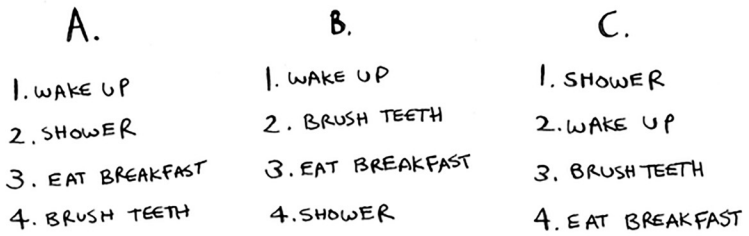
On the other hand, showering doesn't depend on brushing my teeth, because I can shower before I brush my teeth. From this graph, you can make a list of the order in which I need to do my morning routine:

1. Wake up.
2. Shower.
3. Brush teeth.
4. Eat breakfast.

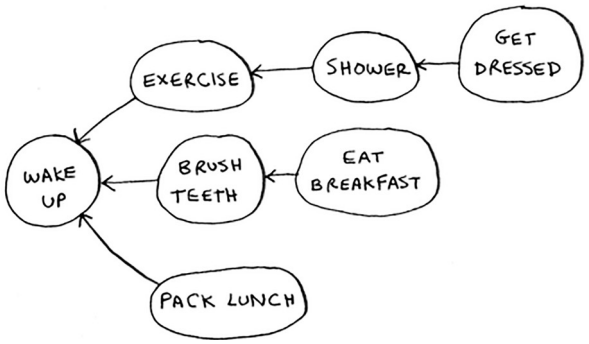
Note that "shower" can be moved around, so this list is also valid:

1. Wake up.
2. Brush teeth.
3. Shower.
4. Eat breakfast.

6.3 For these three lists, mark whether each one is valid or invalid.

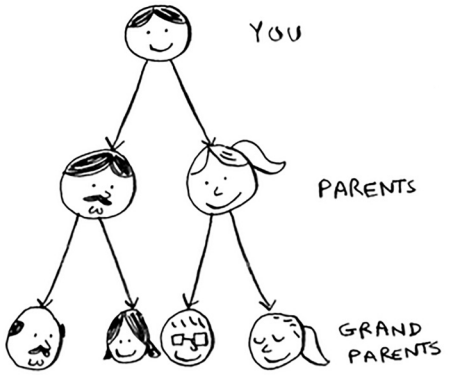


6.4 Here's a larger graph. Make a valid list for this graph.

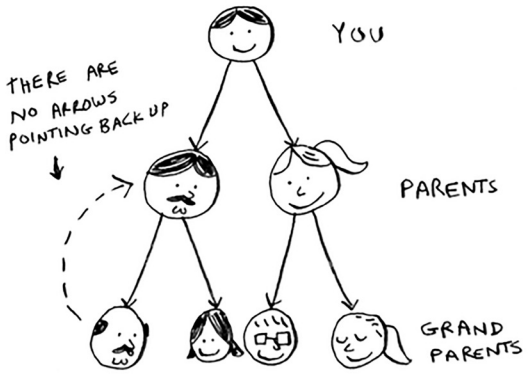


You could say that this list is sorted, in a way. If task A depends on task B, task A shows up later in the list. This is called a *topological sort*, and it's a way to make an ordered list out of a graph. Suppose you're planning a wedding and have a large graph full of tasks to do—and you're not sure where to start. You could *topologically sort* the graph and get a list of tasks to do, in order.

Suppose you have a family tree.

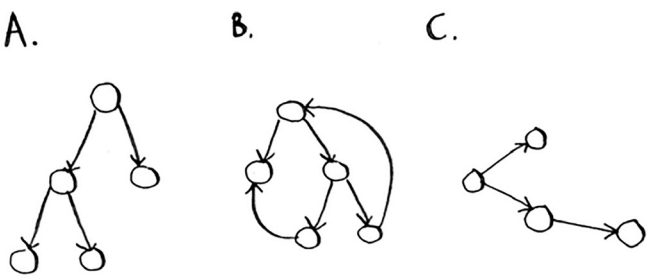


This is a graph, because you have nodes (the people) and edges. The edges point to the nodes' parents. But all the edges go down—it wouldn't make sense for a family tree to have an edge pointing back up! That would be meaningless—your dad can't be your grandfather's dad!



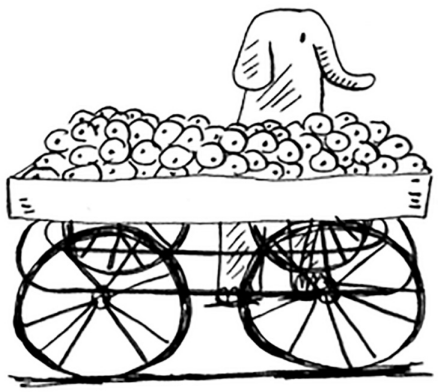
This is called a *tree*. A tree is a special type of graph, where no edges ever point back.

6.5 Which of the following graphs are also trees?



Recap

- Breadth-first search tells you if there's a path from A to B.
- If there's a path, breadth-first search will find the shortest path.
- If you have a problem like "find the shortest X," try modeling your problem as a graph, and use breadth-first search to solve.
- A directed graph has arrows, and the relationship follows the direction of the arrow (rama -> adit means "rama owes adit money").
- Undirected graphs don't have arrows, and the relationship goes both ways (ross - rachel means "ross dated rachel and rachel dated ross").
- Queues are FIFO (First In, First Out).
- Stacks are LIFO (Last In, First Out).
- You need to check people in the order they were added to the search list, so the search list needs to be a queue. Otherwise, you won't get the shortest path.
- Once you check someone, make sure you don't check them again. Otherwise, you might end up in an infinite loop.

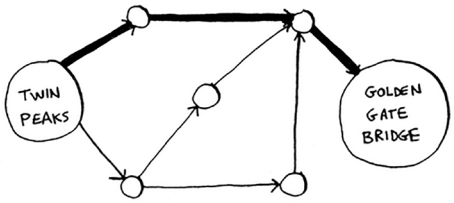


Dijkstra's algorithm 7

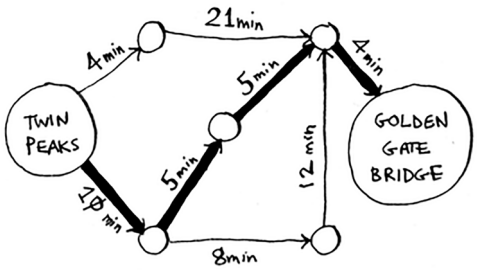
In this chapter

- We continue the discussion of graphs, and you learn about weighted graphs: a way to assign more or less weight to some edges.
- You learn Dijkstra's algorithm, which lets you answer "What's the shortest path to X?" for weighted graphs.
- You learn about cycles in graphs, where Dijkstra's algorithm doesn't work.

In the last chapter, you figured out a way to get from point A to point B.



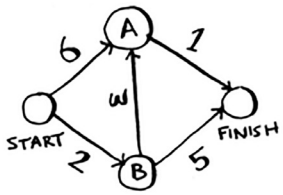
It's not necessarily the fastest path. It's the shortest path, because it has the least number of segments (three segments). But suppose you add travel times to those segments. Now you see that there's a faster path.



You used breadth-first search in the last chapter. Breadth-first search will find you the path with the fewest segments (the first graph shown here). What if you want the fastest path instead (the second graph)? You can do that *fastest* with a different algorithm called *Dijkstra's algorithm*.

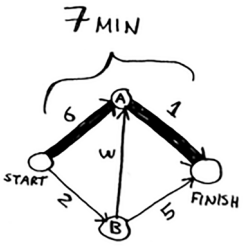
Working with Dijkstra's algorithm

Let's see how it works with this graph.



Each segment has a travel time in minutes. You'll use Dijkstra's algorithm to go from start to finish in the shortest possible time.

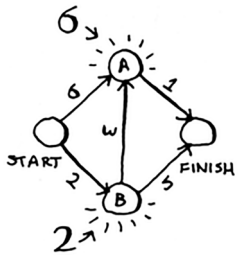
If you ran breadth-first search on this graph, you'd get this shortest path.



But that path takes 7 minutes. Let's see if you can find a path that takes less time! There are four steps to Dijkstra's algorithm:

1. Find the "cheapest" node. This is the node you can get to in the least amount of time.
2. Update the costs of the neighbors of this node. I'll explain what I mean by this shortly.
3. Repeat until you've done this for every node in the graph.
4. Calculate the final path.

Step 1: Find the cheapest node. You're standing at the start, wondering if you should go to node A or node B. How long does it take to get to each node?

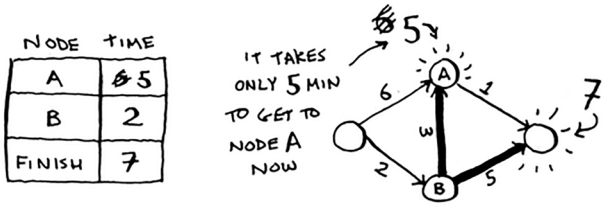


It takes 6 minutes to get to node A and 2 minutes to get to node B. The rest of the nodes, you don't know yet.

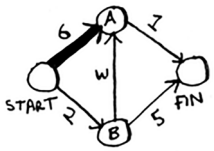
Because you don't know how long it takes to get to the finish yet, you put down infinity (you'll see why soon). Node B is the closest node … it's 2 minutes away.

NODE	TIME TO NODE
A	6
B	2
FINISH	∞

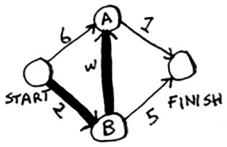
Step 2: Calculate how long it takes to get to all of node B's neighbors *by following an edge from B.*



Hey, you just found a shorter path to node A! It used to take 6 minutes to get to node A.



But if you go through node B, there's a path that only takes 5 minutes!

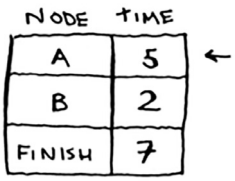


When you find a shorter path for a neighbor of B, update its cost. In this case, you found

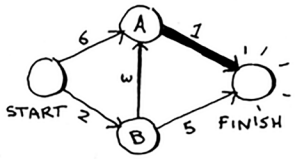
- A shorter path to A (down from 6 minutes to 5 minutes)
- A shorter path to the finish (down from infinity to 7 minutes)

Step 3: Repeat!

Step 1 again: Find the node that takes the least amount of time to get to. You're done with node B, so node A has the next smallest time estimate.



Step 2 again: Update the costs for node A's neighbors.



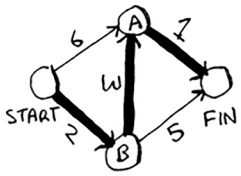
Woo, it takes 6 minutes to get to the finish now!

You've run Dijkstra's algorithm for every node (you don't need to run it for the finish node). At this point, you know

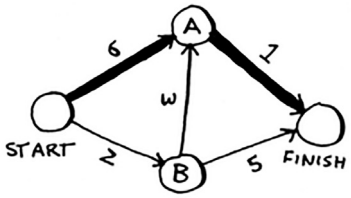
- It takes 2 minutes to get to node B.
- It takes 5 minutes to get to node A.
- It takes 6 minutes to get to the finish.

NODE	TIME
A	5
B	2
FINISH	6

I'll save the last step, calculating the final path, for the next section. For now, I'll just show you what the final path is.

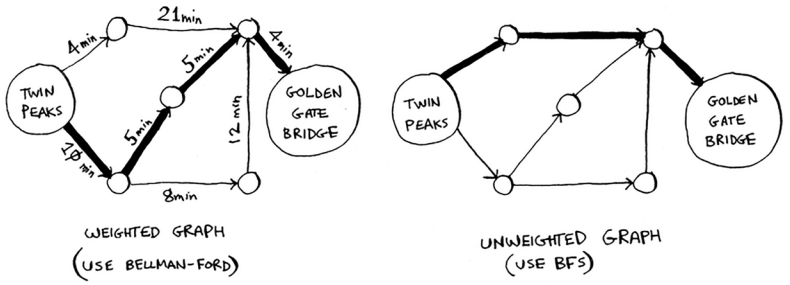


Breadth-first search wouldn't have found this as the shortest path, because it has three segments. And there's a way to get from the start to the finish in two segments.



SHORTEST PATH
WITH BREADTH-FIRST SEARCH

In the last chapter, you used breadth-first search to find the shortest path between two points. Back then, "shortest path" meant the path with the fewest segments. But in Dijkstra's algorithm, you assign a number or weight to each segment. Then Dijkstra's algorithm finds the path with the smallest total weight.



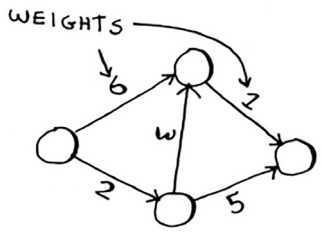
To recap, Dijkstra's algorithm has four steps:

1. Find the cheapest node. This is the node you can get to in the least amount of time.
2. Check whether there's a cheaper path to the neighbors of this node. If so, update their costs.
3. Repeat until you've done this for every node in the graph.
4. Calculate the final path. (Coming up in the next section!)

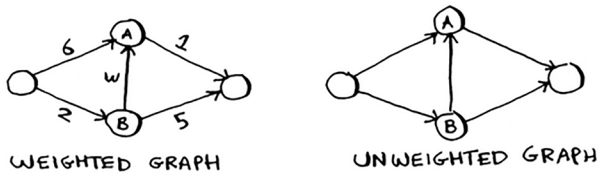
Terminology

I want to show you some more examples of Dijkstra's algorithm in action. But first let me clarify some terminology.

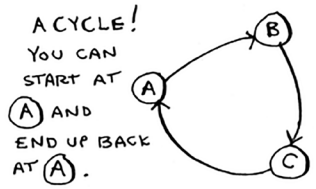
When you work with Dijkstra's algorithm, each edge in the graph has a number associated with it. These are called *weights*.



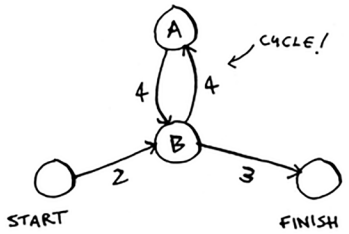
A graph with weights is called a *weighted graph*. A graph without weights is called an *unweighted graph*.



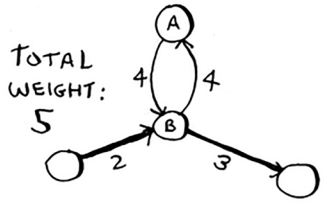
To calculate the shortest path in an unweighted graph, use *breadth-first search*. To calculate the shortest path in a weighted graph, use *Dijkstra's algorithm*. Graphs can also have *cycles*. A cycle looks like this.



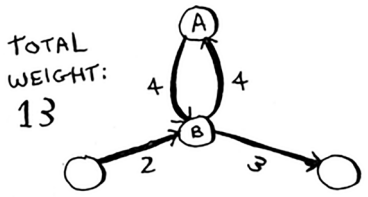
It means you can start at a node, travel around, and end up at the same node. Suppose you're trying to find the shortest path in this graph that has a cycle.



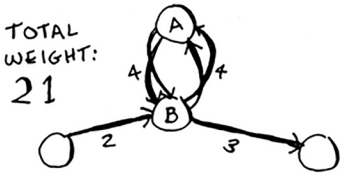
Would it make sense to follow the cycle? Well, you can use the path that avoids the cycle.



Or you can follow the cycle.

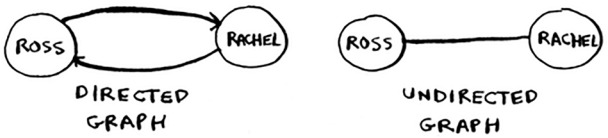


You end up at node A either way, but the cycle adds more weight. You could even follow the cycle twice if you wanted.

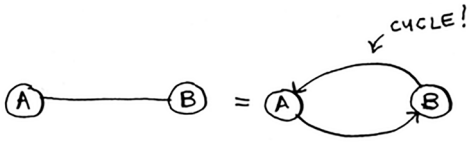


But every time you follow the cycle, you're just adding 8 to the total weight. So following the cycle will never give you the shortest path.

Finally, remember our conversation about directed versus undirected graphs from chapter 6?



An undirected graph means that both nodes point to each other. That's a cycle!



With an undirected graph, each edge adds another cycle. Dijkstra's algorithm only works with *directed acyclic graphs*, called DAGs for short.

Trading for a piano

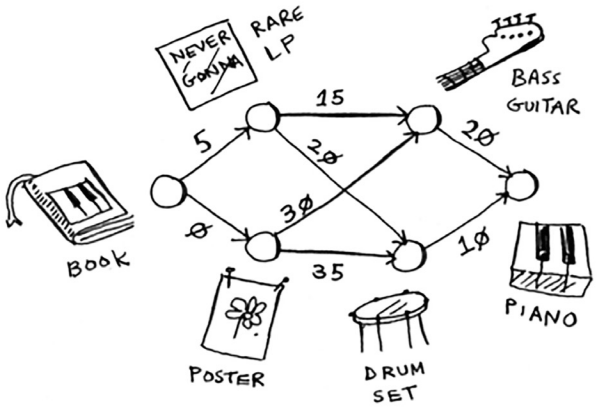
Enough terminology, let's look at another example! This is Rama.

Rama is trying to trade a music book for a piano.

"I'll give you this poster for your book," says Alex. "It's a poster of my favorite band, Destroyer. Or I'll give you this rare LP of Rick Astley for your book and $5 more." "Ooh, I've heard that LP has a really great song," says Amy. "I'll trade you my guitar or drum set for the poster or the LP."

"I've been meaning to get into guitar!" exclaims Beethoven. "Hey, I'll trade you my piano for either of Amy's things."

Perfect! With a little bit of money, Rama can trade his way from a piano book to a real piano. Now he just needs to figure out how to spend the least amount of money to make those trades. Let's graph out what he's been offered.



In this graph, the nodes are all the items Rama can trade for. The weights on the edges are the amount of money he would have to pay to make the trade. So he can trade the poster for the guitar for $30, or trade the LP for the guitar for $15. How is Rama going to figure out the path from the book to the piano where he spends the least dough? Dijkstra's algorithm to the rescue! Remember, Dijkstra's algorithm has four steps. In this example, you'll do all four steps, so you'll calculate the final path at the end, too.

NODE	COST
LP	5
POSTER	∅
GUITAR	∞
DRUMS	∞
PIANO	∞

WE HAVEN'T REACHED THESE NODES FROM THE START YET

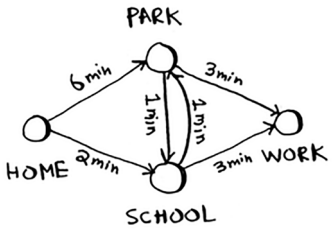
Before you start, you need some setup. Make a table of the cost for each node. The cost of a node is how expensive it is to get to.

You'll keep updating this table as the algorithm goes on. To calculate the final path, you also need a *parent* column on this table.

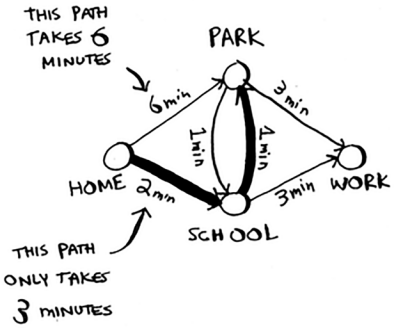
NODE	PARENT
LP	BOOK
POSTER	BOOK
GUITAR	—
DRUMS	—
PIANO	—

I'll show you how this column works soon. Let's start the algorithm.

Step 1: Find the cheapest node. In this case, the poster is the cheapest trade, at $0. Is there a cheaper way to trade for the poster? This is a really important point, so think about it. Can you see a series of trades that will get Rama the poster for less than $0? Read on when you're ready. Answer: No. *Because the poster is the cheapest node Rama can get to, there's no way to make it any cheaper.* Here's a different way to look at it. Suppose you're traveling from home to work.



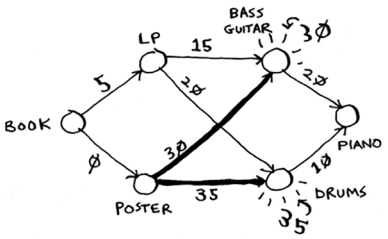
If you take the path toward the school, that takes 2 minutes. If you take the path toward the park, that takes 6 minutes. Is there any way you can take the path toward the park, and end up at the school, in less than 2 minutes? It's impossible, because it takes longer than 2 minutes just to get to the park. On the other hand, can you find a faster path to the park? Yup.



This is the key idea behind Dijkstra's algorithm: *Look at the cheapest node on your graph. There is no cheaper way to get to this node!*

Back to the music example. The poster is the cheapest trade.

Step 2: Figure out how long it takes to get to its neighbors (the cost).



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
POSTER	GUITAR	~~30~~
POSTER	DRUMS	~~35~~
—	PIANO	∞

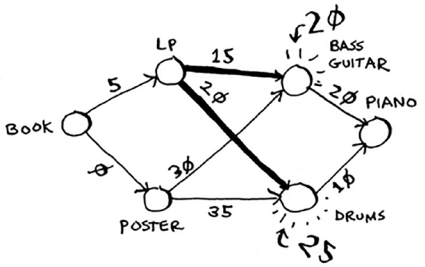
You have prices for the bass guitar and the drum set in the table. Their value was set when you went through the poster, so the poster gets set as their parent. That means, to get to the bass guitar, you follow the edge from the poster, and the same for the drums.

WE GO FROM "POSTER" TO GET TO THESE NODES

PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
POSTER	GUITAR	30
POSTER	DRUMS	35
—	PIANO	∞

Step 1 again: The LP is the next cheapest node at $5.

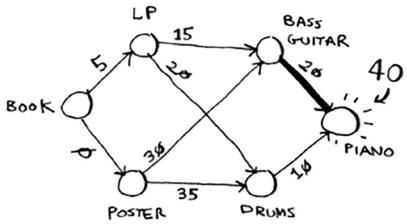
Step 2 again: Update the values of all of its neighbors.



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	~~30~~ 20
LP	DRUMS	~~35~~ 25
—	PIANO	∞

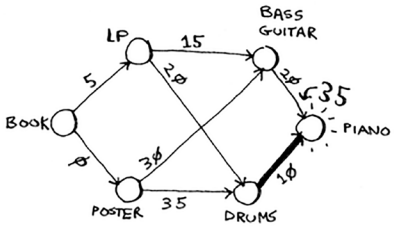
Hey, you updated the price of both the drums and the guitar! That means it's cheaper to get to the drums and guitar by following the edge from the LP. So you set the LP as the new parent for both instruments.

The bass guitar is the next cheapest item. Update its neighbors.



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	20
LP	DRUMS	25
GUITAR	PIANO	40

Ok, you finally have a price for the piano, by trading the guitar for the piano. So you set the guitar as the parent. Finally, the last node, the drum set.



PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	20
LP	DRUMS	25
DRUMS	PIANO	35

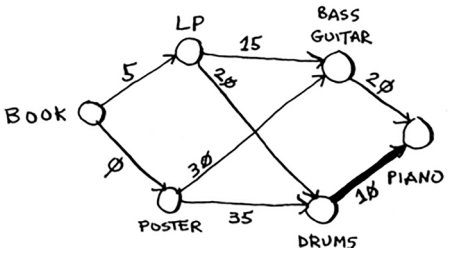
Rama can get the piano even cheaper by trading the drum set for the piano instead. *So the cheapest set of trades will cost Rama $35.*

Now, as I promised, you need to figure out the path. So far, you know that the shortest path costs $35, but how do you figure out the path? To start with, look at the parent for *piano*.

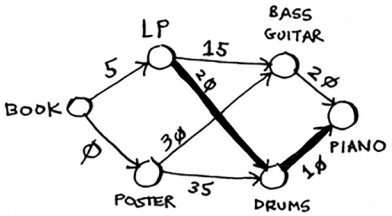
PARENT	NODE
BOOK	LP
BOOK	POSTER
LP	GUITAR
LP	DRUMS
→ DRUMS	PIANO

The piano has drums as its parent. That means Rama trades the drums for the piano. So you follow this edge.

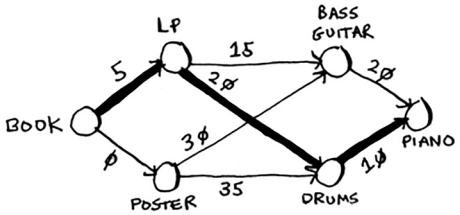
Let's see how you'd follow the edges. *Piano* has *drums* as its parent.



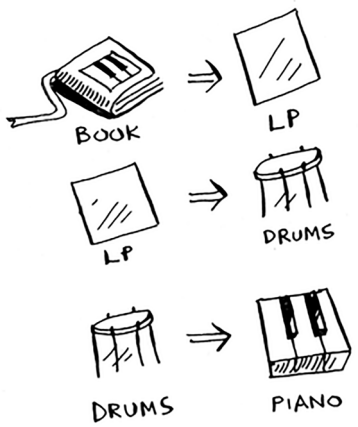
And *drums* has the LP as its parent.



So Rama will trade the LP for the drums. And of course, he'll trade the book for the LP. By following the parents backward, you now have the complete path.

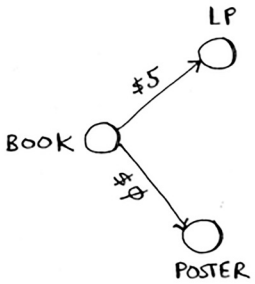


Here's the series of trades Rama needs to make.



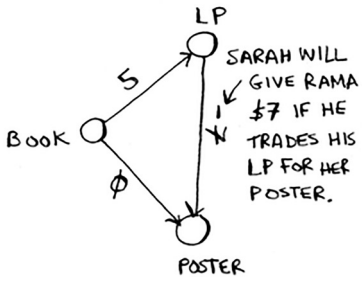
So far, I've been using the term *shortest path* pretty literally: calculating the shortest path between two locations or between two people. I hope this example showed you that the shortest path doesn't have to be about physical distance. It can be about minimizing something. In this case, Rama wanted to minimize the amount of money he spent. Thanks, Dijkstra!

Negative-weight edges

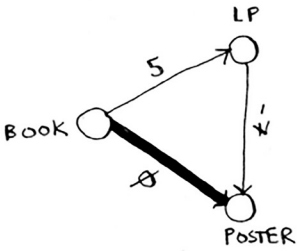


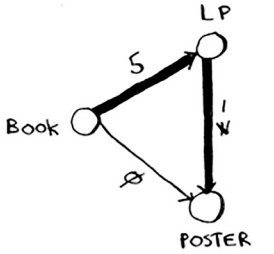
In the trading example, Alex offered to trade the book for two items.

Suppose Sarah offers to trade the LP for the poster, and *she'll give Rama an additional $7*. It doesn't cost Rama anything to make this trade; instead, he gets $7 back. How would you show this on the graph?

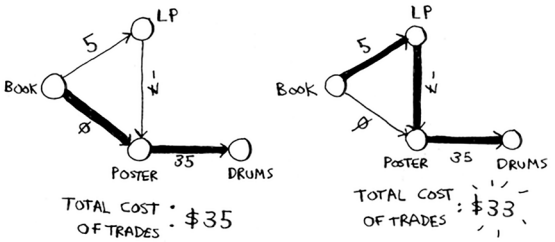


The edge from the LP to the poster has a negative weight! Rama gets $7 back if he makes that trade. Now Rama has two ways to get to the poster.

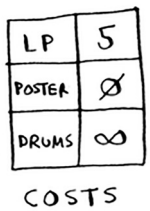




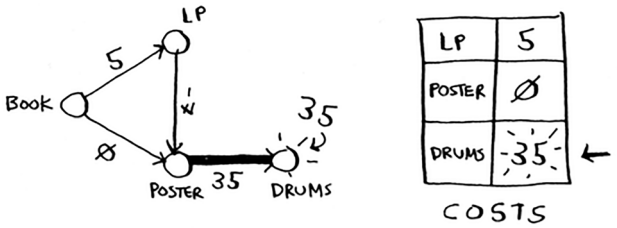
So it makes sense to do the second trade—Rama gets $2 back that way! Now, if you remember, Rama can trade the poster for the drums. There are two paths he could take.



The second path costs him $2 less, so he should take that path, right? Well, guess what? If you run Dijkstra's algorithm on this graph, Rama will take the wrong path. He'll take the longer path. *You can't use Dijkstra's algorithm if you have negative-weight edges.* Negative-weight edges break the algorithm. Let's see what happens when you run Dijkstra's algorithm on this. First, make the table of costs.

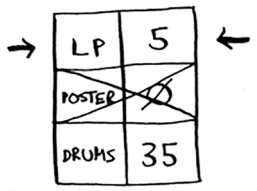


Next, find the lowest-cost node, and update the costs for its neighbors. In this case, the poster is the lowest-cost node. So, according to Dijkstra's algorithm, *there is no cheaper way to get to the poster than paying $0* (you know that's wrong!). Anyway, let's update the costs for its neighbors.

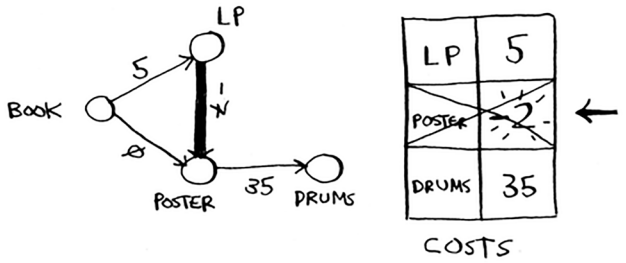


Ok, the drums have a cost of $35 now.

Let's get the next-cheapest node that hasn't already been processed.



Update the costs for its neighbors.

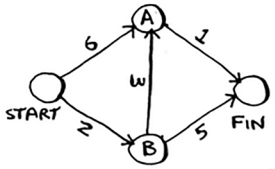


You already processed the poster node, but you're updating the cost for it. This is a big red flag. Once you process a node, it means there's no cheaper way to get to that node. But you just found a cheaper way to the poster! Drums doesn't have any neighbors, so that's the end of the algorithm. Here are the final costs.

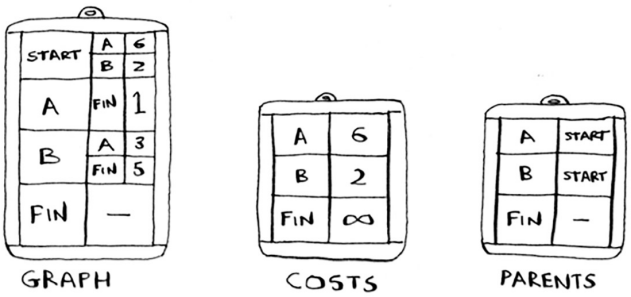
It costs $35 to get to the drums. You know that there's a path that costs only $33, but Dijkstra's algorithm didn't find it. Dijkstra's algorithm assumed that because you were processing the poster node, there was no faster way to get to that node. That assumption only works if you have no negative-weight edges. So you *can't use negative-weight edges with Dijkstra's algorithm.* If you want to find the shortest path in a graph that has negative-weight edges, there's an algorithm for that! It's called the *Bellman-Ford algorithm*. Bellman-Ford is out of the scope of this book, but you can find some great explanations online.

Implementation

Let's see how to implement Dijkstra's algorithm in code. Here's the graph I'll use for the example.



To code this example, you'll need three hash tables.



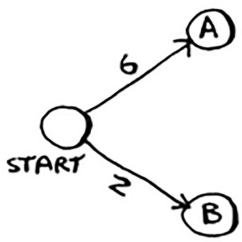
You'll update the costs and parents hash tables as the algorithm progresses. First, you need to implement the graph. You'll use a hash table like you did in chapter 6:

```
graph = {}
```

In the last chapter, you stored all the neighbors of a node in the hash table, like this:

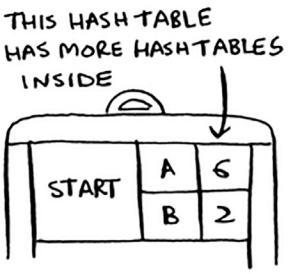
```
graph["you"] = ["alice", "bob", "claire"]
```

But this time, you need to store the neighbors *and* the cost for getting to that neighbor. For example, Start has two neighbors, A and B.



How do you represent the weights of those edges? Why not just use another hash table?

```
graph["start"] = {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2
```



So `graph["start"]` is a hash table. You can get all the neighbors for Start like this:

```
>>> print graph["start"].keys()
["a", "b"]
```

There's an edge from Start to A and an edge from Start to B. What if you want to find the weights of those edges?

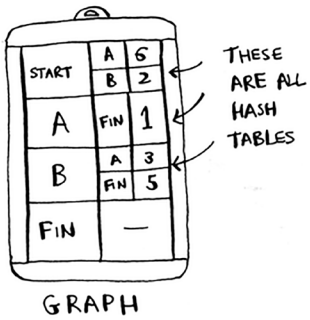
```
>>> print graph["start"]["a"]
2
>>> print graph["start"]["b"]
6
```

Let's add the rest of the nodes and their neighbors to the graph:

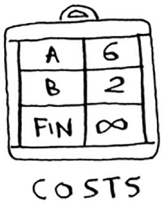
```
graph["a"] = {}
graph["a"]["fin"] = 1

graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5

graph["fin"] = {}
```

The finish node doesn't have any neighbors.

The full graph hash table looks like this.



Next you need a hash table to store the costs for each node.



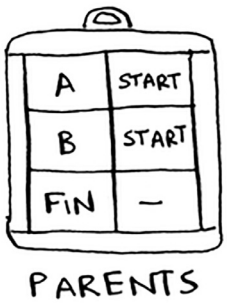
The *cost* of a node is how long it takes to get to that node from the start. You know it takes 2 minutes from Start to node B. You know it takes 6 minutes to get to node A (although you may find a path that takes less time). You don't know how long it takes to get to the finish. If you don't know the cost yet, you put down infinity. Can you represent *infinity* in Python? Turns out, you can:

```
infinity = float("inf")
```

Here's the code to make the costs table:

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

You also need another hash table for the parents:



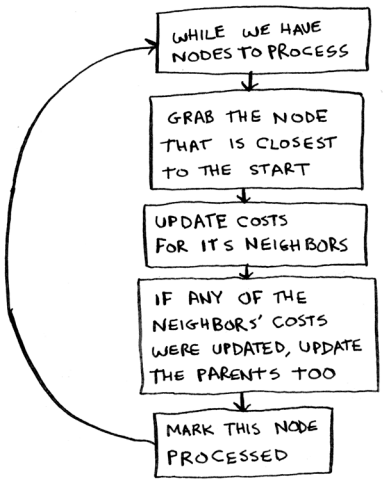
Here's the code to make the hash table for the parents:

```
parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["fin"] = None
```

Finally, you need an array to keep track of all the nodes you've already processed, because you don't need to process a node more than once:

```
processed = []
```

That's all the setup. Now let's look at the algorithm.



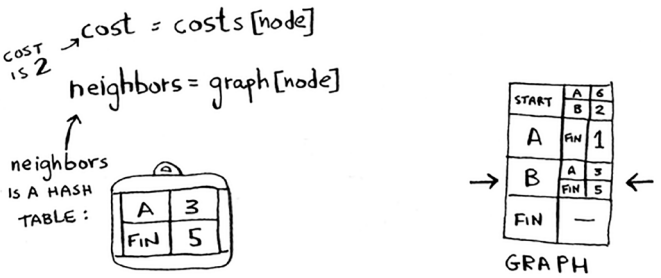
I'll show you the code first and then walk through it. Here's the code:

```
node = find_lowest_cost_node(costs)   <- Find the lowest-cost node that you haven't processed yet.
while node is not None:               <- If you've processed all the nodes, this while loop is done.
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():        <- Go through all the neighbors of this node.
        new_cost = cost + neighbors[n]    <- If it's cheaper to get to this neighbor
        if costs[n] > new_cost:           <- by going through this node ...
            costs[n] = new_cost           <- ... update the cost for this node.
            parents[n] = node             <- This node becomes the new parent for this neighbor.
    processed.append(node)            <- Mark the node as processed.
    node = find_lowest_cost_node(costs)   <- Find the next node to process, and loop.
```

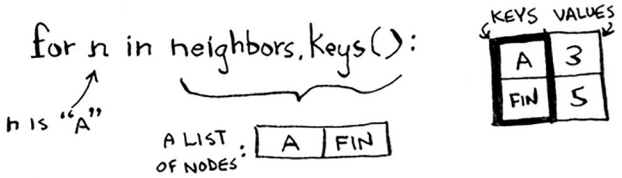
That's Dijkstra's algorithm in Python! I'll show you the code for the function later. First, let's see this `find_lowest_cost_node` algorithm code in action.

Find the node with the lowest cost.

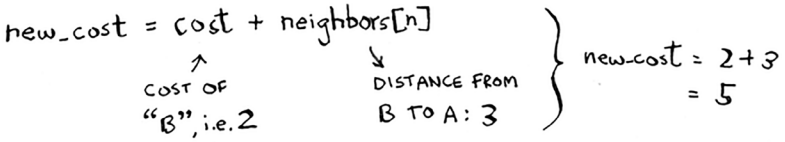
Get the cost and neighbors of that node.



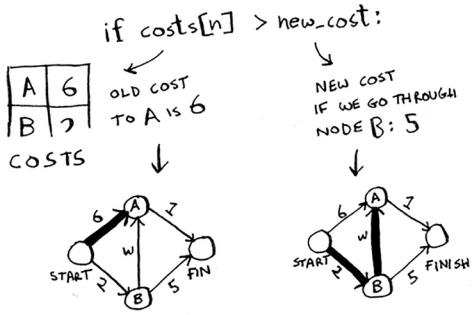
Loop through the neighbors.



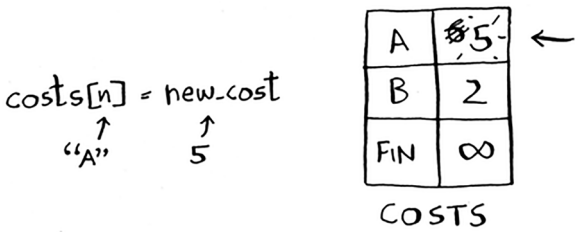
Each node has a cost. The cost is how long it takes to get to that node from the start. Here, you're calculating how long it would take to get to node A if you went Start > node B > node A, instead of Start > node A.



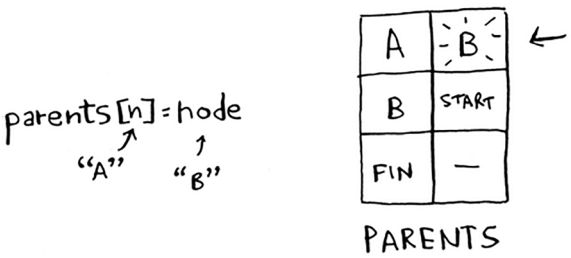
Let's compare those costs.



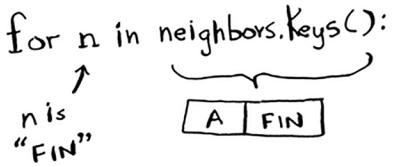
You found a shorter path to node A! Update the cost.



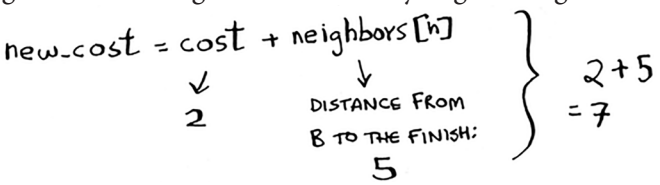
The new path goes through node B, so set B as the new parent.



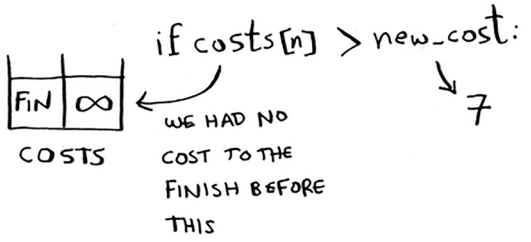
Ok, you're back at the top of the loop. The next neighbor `for` is the Finish node.



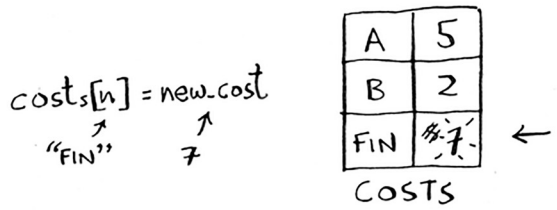
How long does it take to get to the finish if you go through node B?

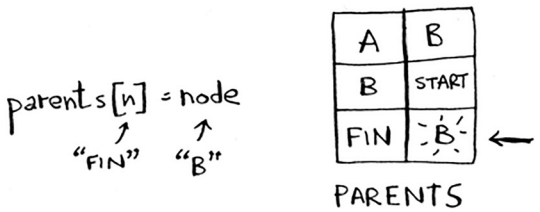


It takes 7 minutes. The previous cost was infinity minutes, and 7 minutes is less than that.

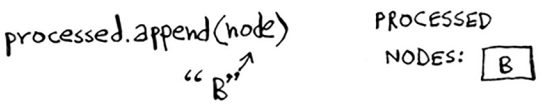


Set the new cost and the new parent for the Finish node.

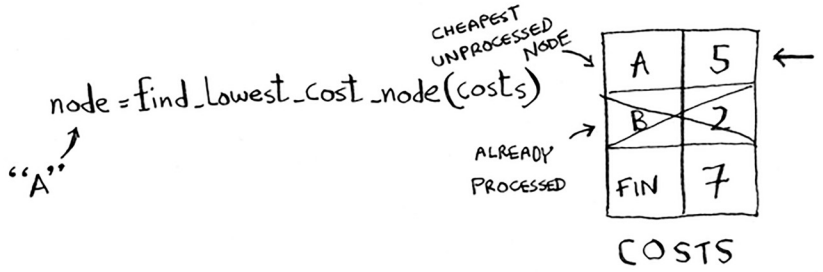




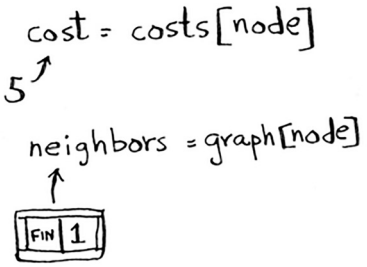
Ok, you updated the costs for all the neighbors of node B. Mark it as processed.



Find the next node to process.

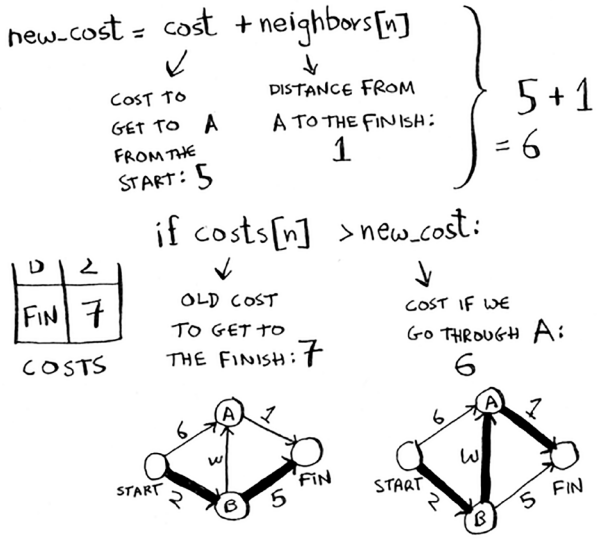


Get the cost and neighbors for node A.

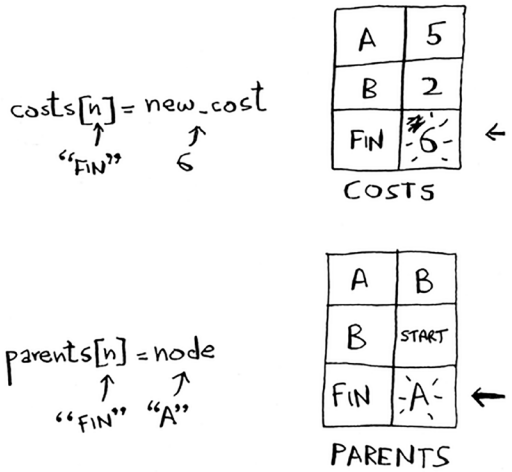


Node A only has one neighbor: the Finish node.

Currently it takes 7 minutes to get to the Finish node. How long would it take to get there if you went through node A?



It's faster to get to Finish from node A! Let's update the cost and parent.

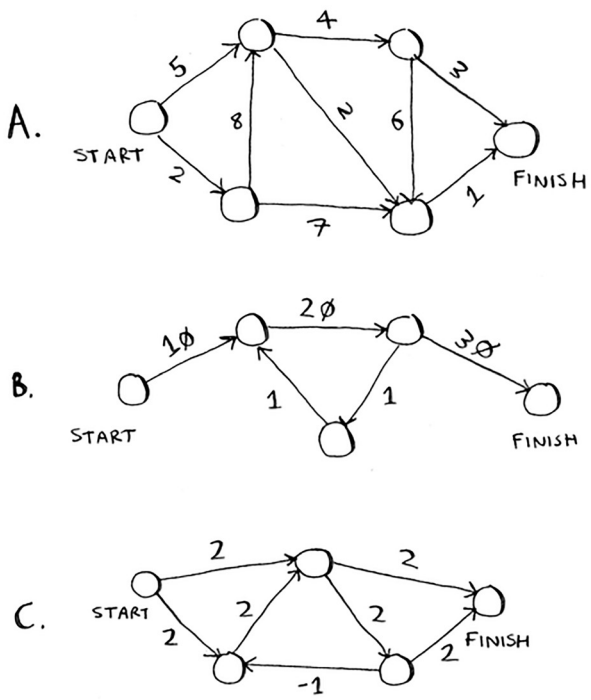


Once you've processed all the nodes, the algorithm is over. I hope the walkthrough helped you understand the algorithm a little better. Finding the lowest-cost node is pretty easy with the `find_lowest_cost_node` function. Here it is in code:

```
def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None
    for node in costs:                                  # Go through each node.
        cost = costs[node]
        if cost < lowest_cost and node not in processed:  # If it's the lowest cost so far and hasn't been processed yet ...
            lowest_cost = cost                          # ... set it as the new lowest-cost node.
            lowest_cost_node = node
    return lowest_cost_node
```

EXERCISE

7.1 In each of these graphs, what is the weight of the shortest path from start to finish?



Recap

- Breadth-first search is used to calculate the shortest path for an unweighted graph.
- Dijkstra's algorithm is used to calculate the shortest path for a weighted graph.
- Dijkstra's algorithm works when all the weights are positive.
- If you have negative weights, use the Bellman-Ford algorithm.