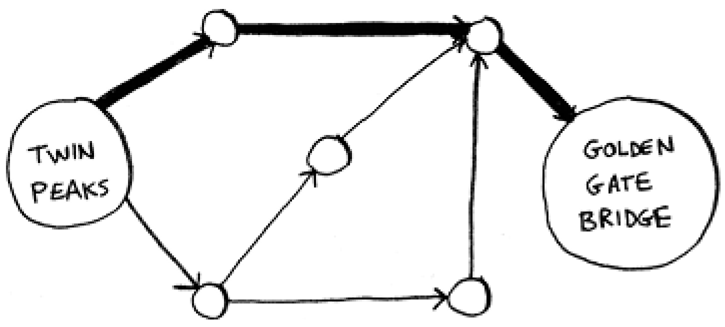
9 Dijkstra's algorithm

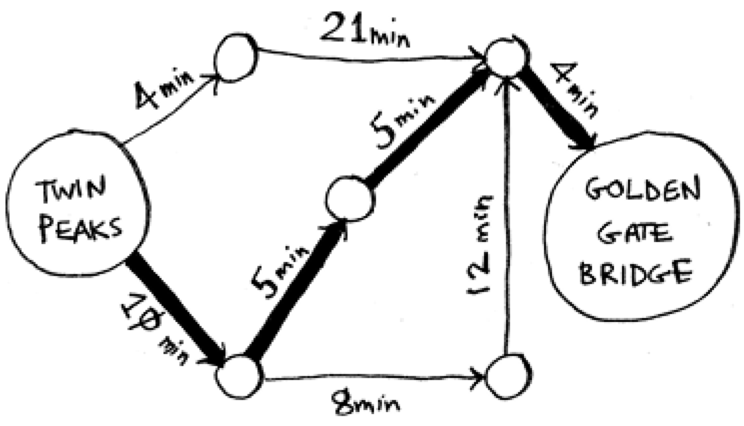
In this chapter

- 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 negative-weight edges in graphs, where Dijkstra's algorithm doesn't work.

In chapter 6, you figured out a way to get from point A to point B.



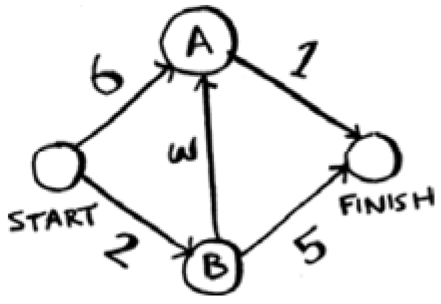
It's not necessarily the fastest path. It's the shortest path because it has the least number of segments (three segments). But suppose you add travel times to those segments. Now you see that there's a faster path.



You used breadth-first search in chapter 6. Breadth-first search will find 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.*

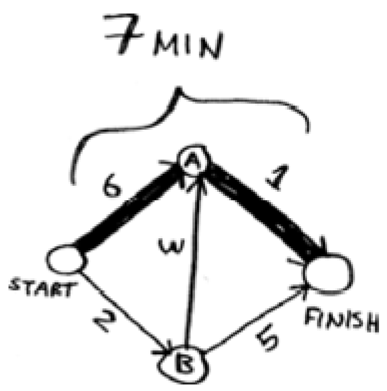
Working with Dijkstra's algorithm

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.

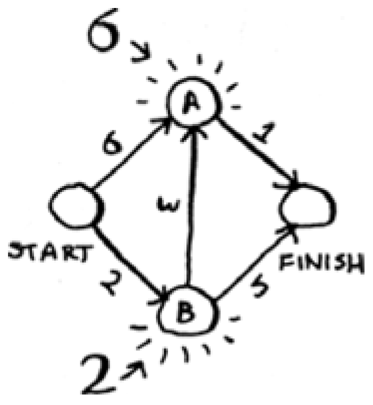
If you ran breadth-first search on this graph, you'd get this shortest path.



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:

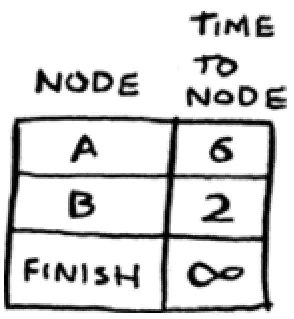
1. Find the "cheapest" node. This is the node you can get to in the least amount of time.
2. Update the costs of the out-neighbors of this node. I'll explain what I mean by this shortly.
3. Repeat until you've done this for every node in the graph.
4. Calculate the final path.

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



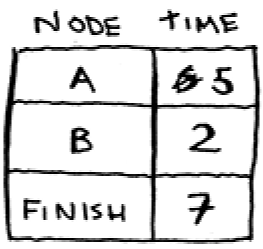
It takes 6 minutes to get to node A and 2 minutes to get to node B. The rest of the nodes, you don't know yet.

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

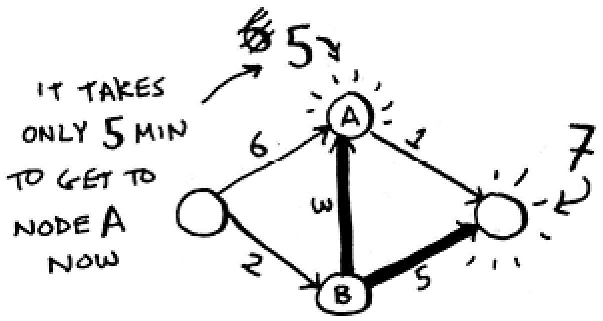


NODE	TIME TO NODE
A	6
B	2
FINISH	∞

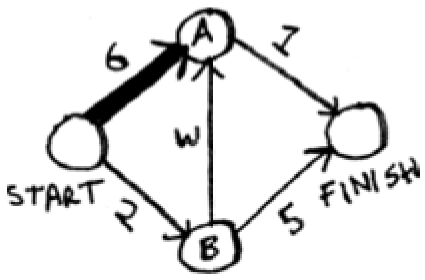
Step 2: Calculate how long it takes to get to all of node B's out-neighbors *by following an edge from B.*



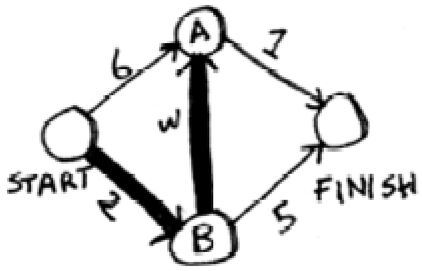
NODE	TIME
A	~~6~~ 5
B	2
FINISH	7



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!

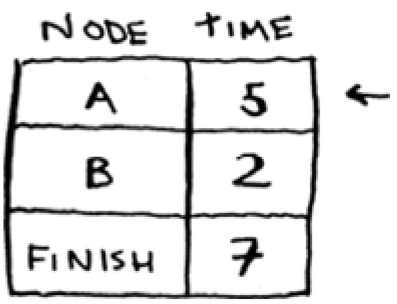


When you find a shorter path for a neighbor of B, update its cost. In this case, you found

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

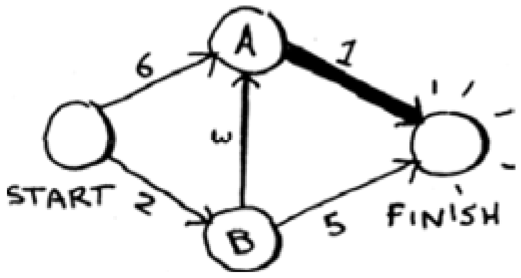
Step 3: Repeat.

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.



NODE	TIME
A	5
B	2
FINISH	7

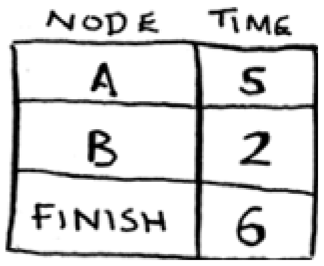
Step 2 again: Update the costs for node A’s out-neighbors.



Woo, it takes 6 minutes to get to Finish now!

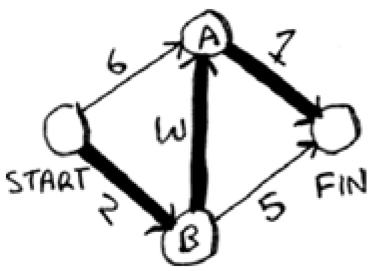
You’ve run Dijkstra’s algorithm for every node (you don’t need to run it for the Finish node). At this point, you know

- 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 Finish.

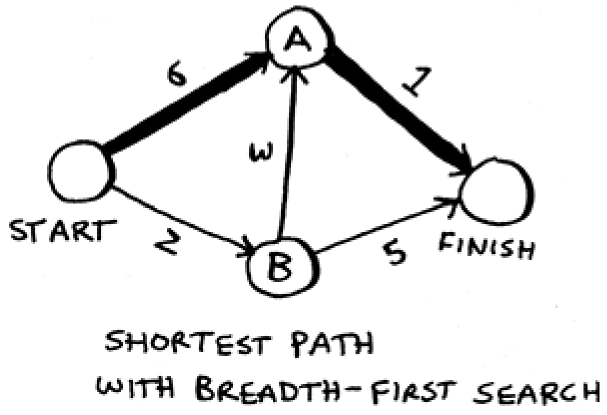


NODE	TIME
A	5
B	2
FINISH	6

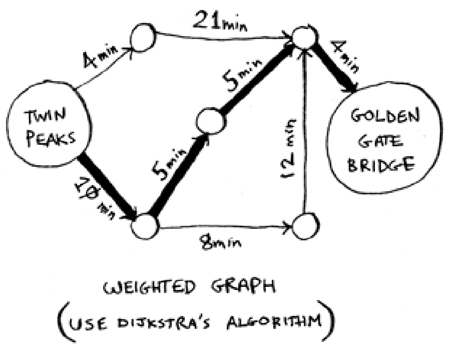
I’ll save the last step, calculating the final path, for the next section. For now, I’ll just show you what the final path is.

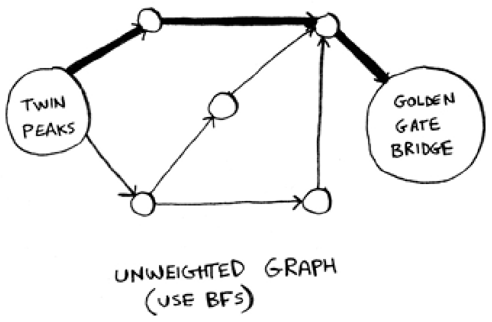


Breadth-first search wouldn’t have found this as the shortest path because it has three segments. And there’s a way to get from Start to Finish in two segments.



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.





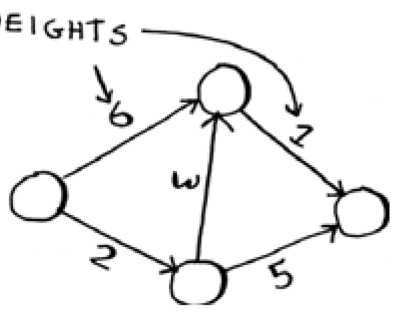
To recap, Dijkstra’s algorithm has four steps:

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 out-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!)

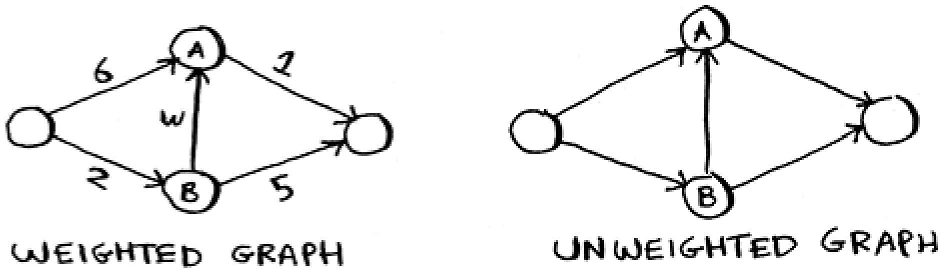
Terminology

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

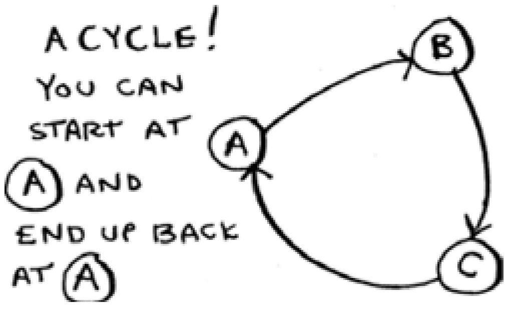
When you work with Dijkstra’s algorithm, each edge in the graph has a number associated with it. These are called *weights.*



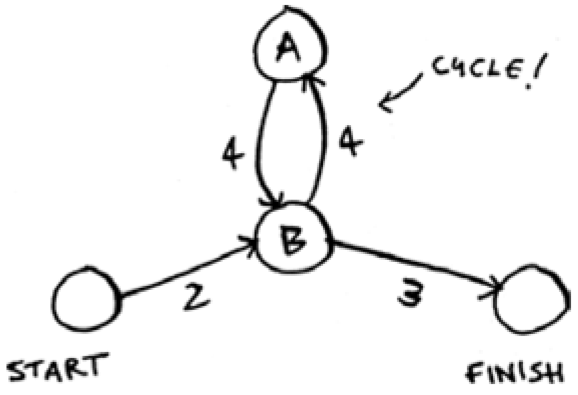
A graph with weights is called a *weighted graph*. A graph without weights is called an *unweighted graph*.



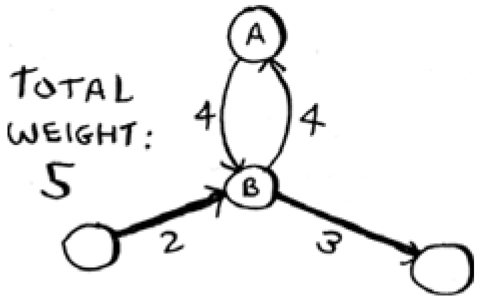
To calculate the shortest path in an unweighted graph, use *breadth-first search*. To calculate the shortest path in a weighted graph, use *Dijkstra's algorithm*. Graphs can also have *cycles*. A cycle looks like this.



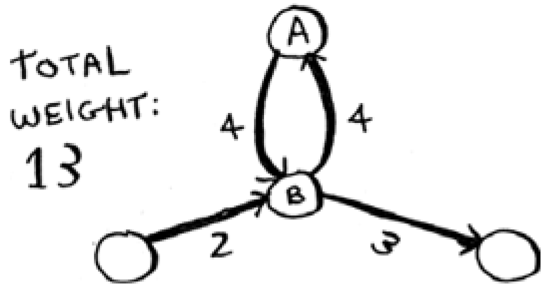
It means you can start at a node, travel around, and end up at the same node. Suppose you're trying to find the shortest path in this graph that has a cycle.



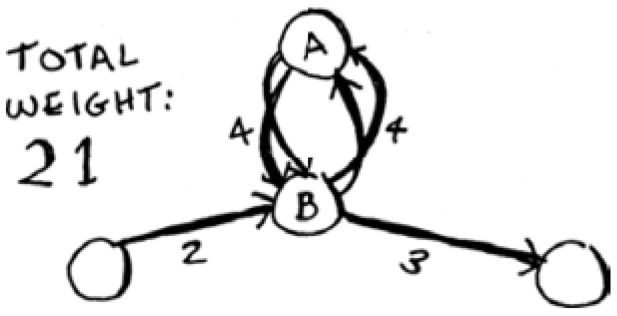
Would it make sense to follow the cycle? Well, you can use the path that avoids the cycle.



Or you can follow the cycle.

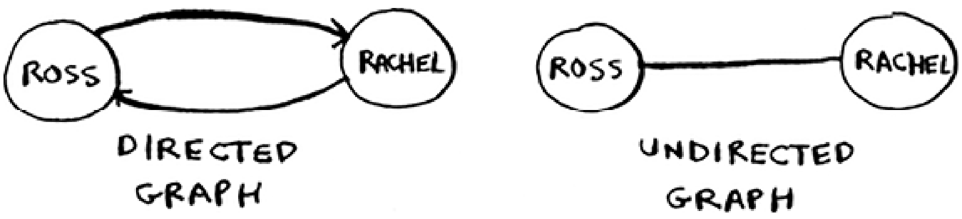


You end up at the target node either way, but the cycle adds more weight. You could even follow the cycle twice if you wanted.

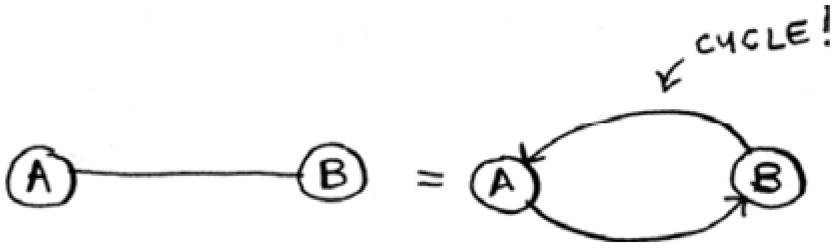


But every time you follow the cycle, you're just adding 8 to the total weight. So following the cycle will never give you the shortest path.

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 on graphs with no cycles, where all the edges are nonnegative. Yes, it's possible for graph edges to have a negative weight! But Dijkstra's algorithm won't work—in that case, you'll need an algorithm called Bellman–Ford. There's a section on negative weight edges coming later in the chapter.

Trading for a piano

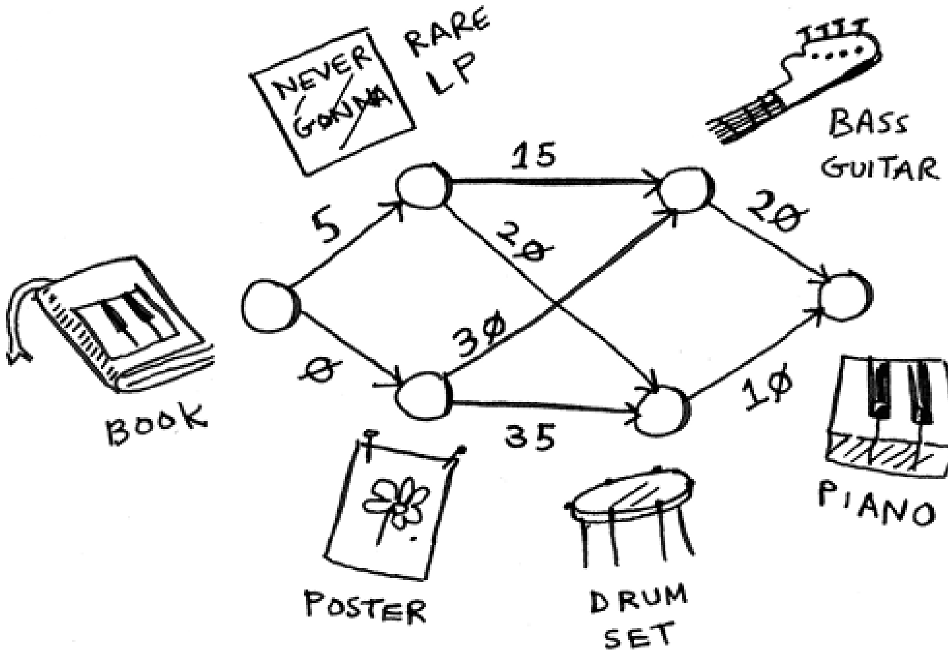
Enough terminology, let's look at another example! This is Rama.

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

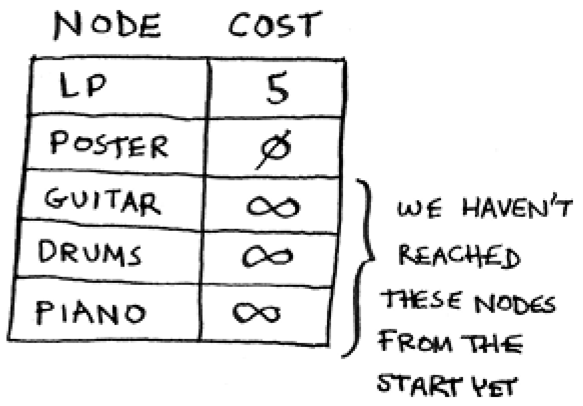
“I’ll give you this poster for your book,” says Alex. “It’s a poster of my favorite band, Destroyer. Or I’ll give you this rare LP of Rick Astley for your book and $5 more.” “Ooh, I’ve heard that LP has a really great song,” says Amy. “I’ll trade you my guitar or drum set for the poster or the LP.”

“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.

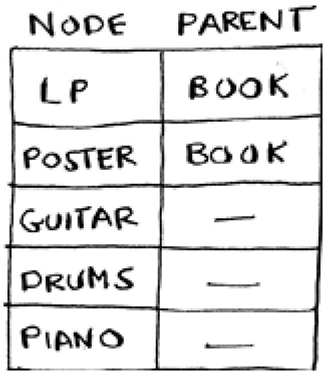


NODE	COST
LP	5
POSTER	Ø
GUITAR	∞
DRUMS	∞
PIANO	∞

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.

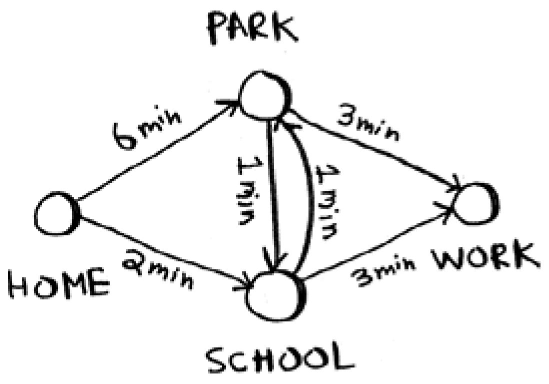
You'll keep updating this table as the algorithm goes on. To calculate the final path, you also need a *parent* column on this table.



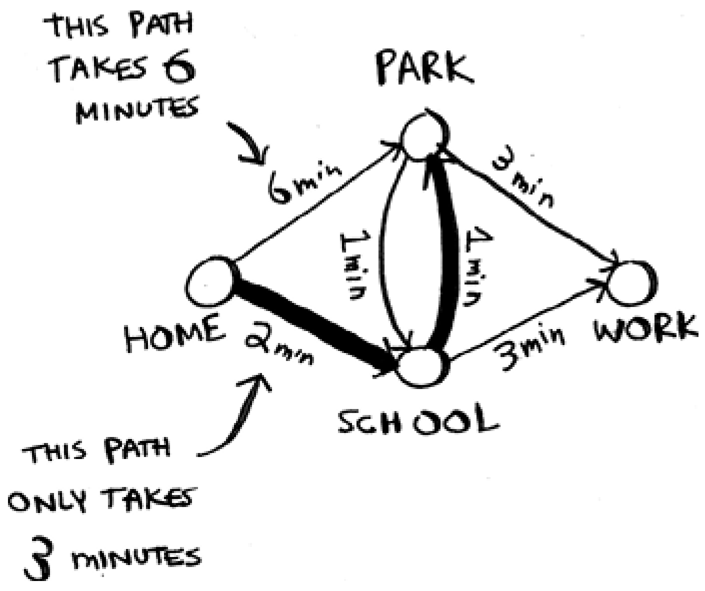
NODE	PARENT
LP	BOOK
POSTER	BOOK
GUITAR	—
DRUMS	—
PIANO	—

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.



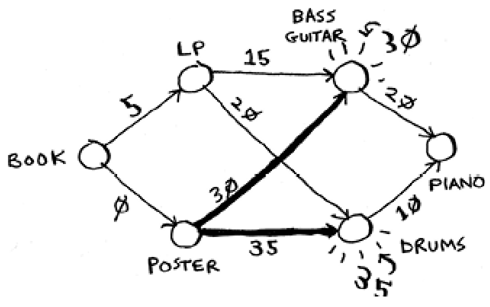
If you take the path toward the school, that takes 2 minutes. If you take the path toward the park, that takes 6 minutes. Is there any way you can take the path toward the park and end up at the school in less than 2 minutes? It’s impossible because it takes longer than 2 minutes just to get to the park.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

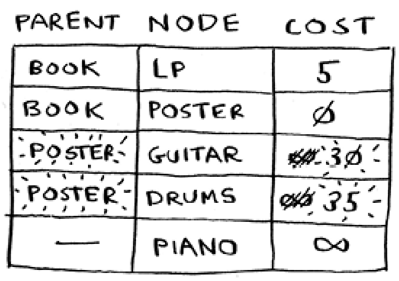


This is the key idea behind Dijkstra's algorithm: look at the cheapest node on your graph. There is no cheaper way to get to this node!

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

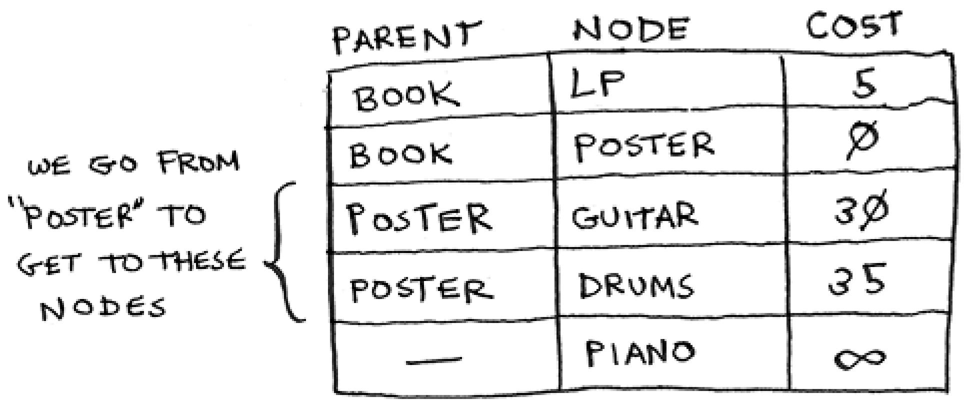
Step 2: Figure out how long it takes to get to its out-neighbors (the cost).





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.

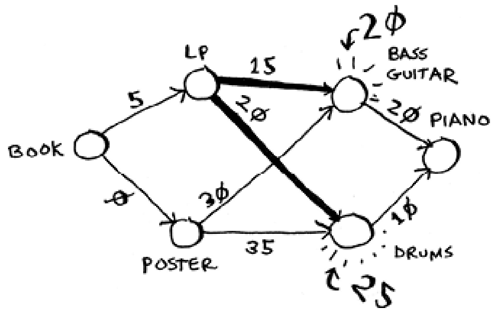


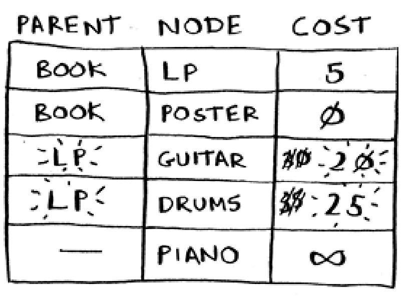
WE GO FROM "POSTER" TO GET TO THESE NODES

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 out-neighbors.

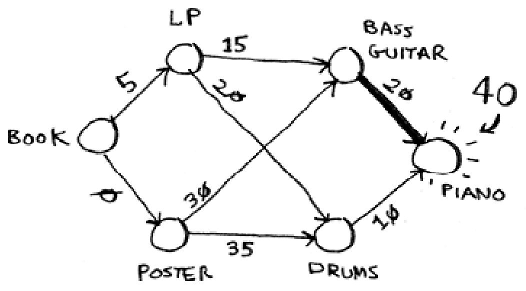


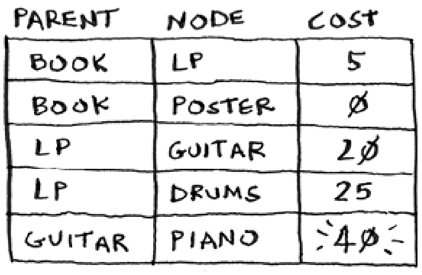


PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	~~30~~ 20
LP	DRUMS	~~35~~ 25
—	PIANO	∞

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.

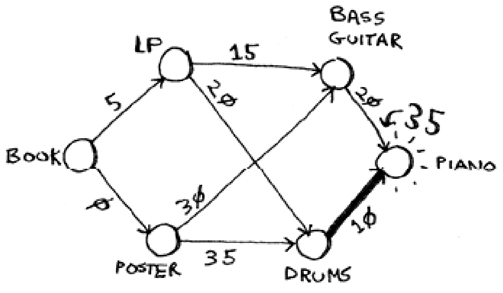
The bass guitar is the next cheapest item. Update its out-neighbors.

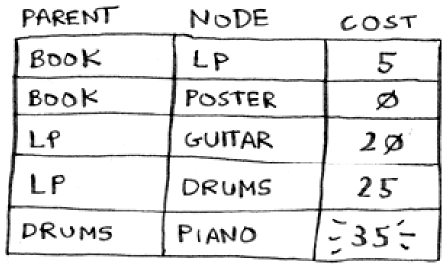




PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	0
LP	GUITAR	20
LP	DRUMS	25
GUITAR	PIANO	40

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.





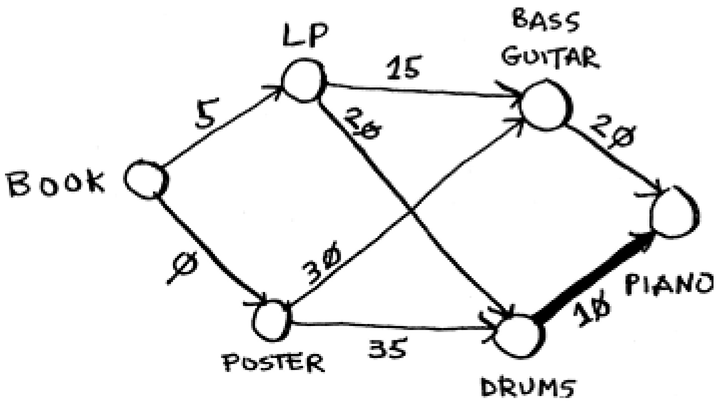
PARENT	NODE	COST
BOOK	LP	5
BOOK	POSTER	Ø
LP	GUITAR	2Ø
LP	DRUMS	25
DRUMS	PIANO	35

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*.

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

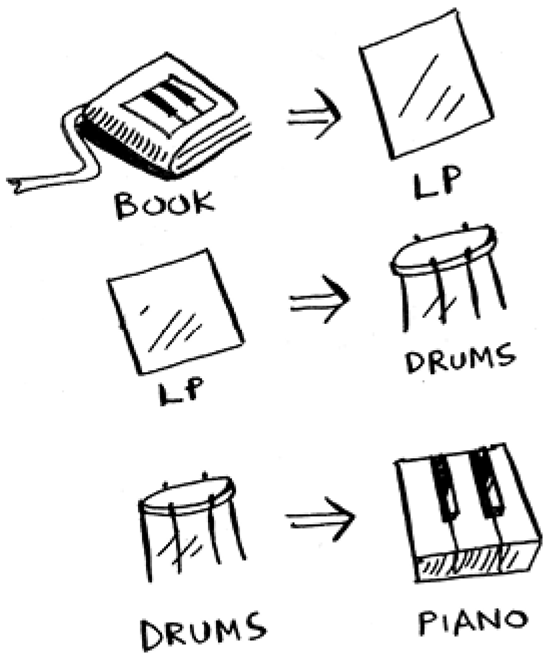
Let’s see how you’d follow the edges. *Piano* has *drums* as its parent.



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 out-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 out-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 out-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 out-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 cheaper 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 out-neighbors of a node in the hash table, like this:

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

But this time, you need to store the out-neighbors *and* the cost for getting to that neighbor. For example, Start has two out-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 out-neighbors for Start like this:

```
>>> print(list(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"])
6
>>> print(graph["start"]["b"])
2
```

Let’s add the rest of the nodes and their out-neighbors to the graph:

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

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

```
graph["fin"] = {}    ①
```

① The Finish node doesn’t have any out-neighbors.

The full graph hash table looks like this.

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

The *cost* of a node is how long it takes to get to that node from 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 Finish. If you don't know the cost yet, you put down infinity. Can you represent *infinity* in Python? Turns out, you can:

```
infinity = math.inf
```

Here's the code to make the costs table:

```
infinity = math.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 a set to keep track of all the nodes you’ve already processed because you don’t need to process a node more than once:

```
processed = set()
```

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)      ①
while node is not None:                  ②
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():           ③
        new_cost = cost + neighbors[n]
        if costs[n] > new_cost:          ④
            costs[n] = new_cost          ⑤
            parents[n] = node            ⑥
    processed.add(node)                  ⑦
    node = find_lowest_cost_node(costs)  ⑧
```

① Finds the lowest-cost node that you haven't processed yet

② If you’ve processed all the nodes, this while loop is done.

③ Goes through all the out-neighbors of this node

④ If it’s cheaper to get to this out-neighbor by going through this node . . .

⑤ . . . updates the cost for the neighbor

⑥ This node becomes the new parent for this out-neighbor.

⑦ Marks the node as processed

⑧ Finds the next node to process and loops

That’s Dijkstra’s algorithm in Python! Let's see this code in action. Then I'll show you the code for the `find_lowest_cost_node` function.

Find the node with the lowest cost.

Get the cost and out-neighbors of that node.

Loop through the out-neighbors.

Each node has a cost. The cost is how long it takes to get to that node from 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 out-neighbor in the `for` loop is the Finish node.

How long does it take to get to 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 out-neighbors of node B. Mark it as processed.

Find the next node to process.

Get the cost and out-neighbors for node A.

Node A only has one out-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 = math.inf
    lowest_cost_node = None
    for node in costs:
        cost = costs[node]
        if cost < lowest_cost and node not in proces
            lowest_cost = cost
            lowest_cost_node = node
    return lowest_cost_node
```

① Goes through each node

② If it’s the lowest cost so far and hasn’t been processed yet . . .

③ . . . sets it as the new lowest-cost node

To find the lowest cost node, we loop through all the nodes each time. There is a more efficient version of this algorithm. It uses a data structure called a priority queue. A priority queue is itself built on top of a different data structure called a heap. If you're curious about priority queues and heaps, check out the section on heaps in the last chapter of the book.

Exercise

9.1 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 nonnegative.
- If you have negative weights, use the Bellman–Ford algorithm.