22

JAVASCRIPT AND PERFORMANCE

Running a computer program on a machine requires bridging the gap between the programming language and the machine's own instruction format. This can be done by writing a program that *interprets* other programs, as we did in Chapter 11, but it's usually done by *compiling* (translating) the program to machine code.

Some languages, such as the C and Rust programming languages, are designed to express roughly those things that the machine is good at. This makes them easy to compile efficiently. JavaScript is designed in a very different way, focusing on simplicity and ease of use. Almost none of the operations it lets you express correspond directly to features of the machine. That makes JavaScript a lot more difficult to compile.

Yet somehow modern JavaScript *engines* (the programs that compile and run JavaScript) do manage to execute programs at an impressive speed. It is possible to write JavaScript programs that are only a few times slower than an equivalent C or Rust program. That may still sound like a big gap, but older JavaScript engines (as well as contemporary implementations of languages with a similar design, such as Python and Ruby) tend to be closer to 100 times slower than C. Compared to these languages, modern JavaScript is strikingly fast—so fast that you'll rarely be forced to switch to another language because of performance problems.

Still, you may need to adjust your code to avoid the slower aspects of the language. As an example of that process, this chapter works through a speed-hungry program and makes it faster. In the process, we'll discuss the way JavaScript engines compile your programs.

Staged Compilation

First, you must understand that JavaScript compilers do not simply compile a program once, the way classical compilers do. Instead, code is compiled and recompiled as needed while the program is running.

With most languages, compiling a big program takes a while. That's usually acceptable, because programs are compiled ahead of time and distributed in compiled form.

For JavaScript, the situation is different. A website might include a large amount of code that is retrieved in text form and must be compiled every time the website is opened. If that process took five minutes, the user wouldn't be happy. A JavaScript compiler must be able to start running a program—even a big program—almost instantaneously.

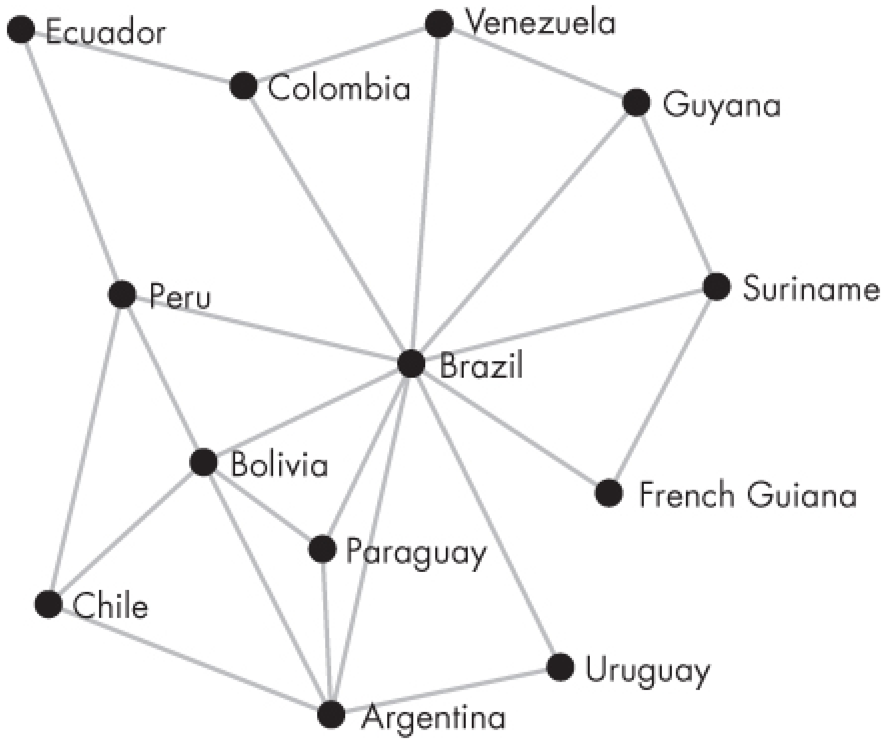
To do this, these compilers have multiple compilation strategies. When a website is first opened, the scripts are first compiled in a cheap, superficial way. This doesn't result in very fast execution, but it allows the script to start quickly. Functions may not be compiled at all until the first time they are called.

In a typical program, most code is run only a handful of times (or not at all). For these parts of the program, the cheap compilation strategy is sufficient—they won't take much time anyway. But functions that are called often or contain loops that do a lot of work have to be treated differently. While running the program, the JavaScript engine observes how often each piece of code runs. When it looks like some code might consume a serious amount of time (this is often called *hot code*), it is recompiled with a more advanced but slower compiler. This compiler performs more *optimizations* to produce faster code. There may even be more than two compilation strategies, with ever more expensive optimizations being applied to *very* hot code.

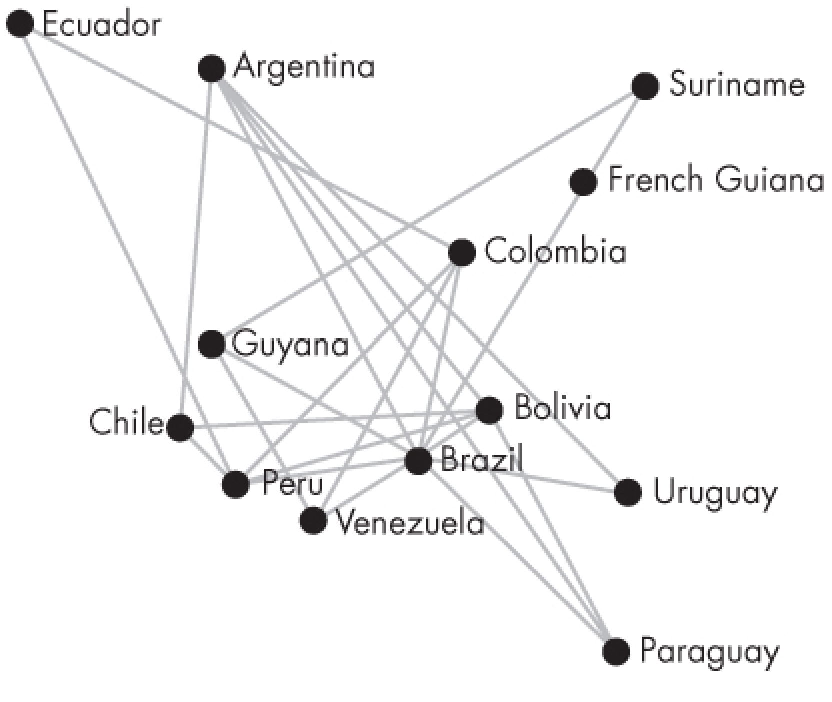
Interleaving running and compiling code means that by the time the clever compiler starts working with a piece of code, it has already been run multiple times. This makes it possible to *observe* the running code and gather information about it. Later in the chapter we'll see how that can allow the compiler to create more efficient code.

Graph Layout

Our example problem for this chapter concerns itself with graphs again. Pictures of graphs can be useful to describe road systems, networks, the way control flows through a computer program, and so on. The following picture shows a graph representing South American countries and territories, with edges between those that share land borders:



Deriving a picture like this from the definition of a graph is called *graph layout*. It involves assigning a place to each node in such a way that connected nodes are near each other, yet the nodes don't crowd into each other. A random layout of the same graph is a lot harder to interpret.



Finding a nice-looking layout for a given graph is a notoriously difficult problem. There is no known solution that reliably does this for arbitrary graphs. Large, densely connected graphs are especially challenging. But for some specific types of graphs, such as *planar* ones (which can be drawn without edges crossing each other), effective approaches exist.

To lay out a small graph that isn't too tangled, we can apply an approach called *force-directed graph layout*. This runs a simplified physics simulation on the nodes of the graph, treating edges as if they are springs and having the nodes themselves repel each other as if electrically charged.

In this chapter we'll implement a force-directed graph layout system and observe its performance. We can run such a simulation by repeatedly computing the forces that act on each node and moving the nodes around in response to those forces. The performance of such a program is important, since it might take a lot of iterations to reach a good-looking layout and each iteration computes a lot of forces.

Defining a Graph

We can represent a graph using a class like this. Each node gets a number, starting from 0, and stores a set of connected nodes.

```
class Graph {
  #nodes = [];

  get size() {
    return this.#nodes.length;
  }

  addNode() {
    let id = this.#nodes.length;
    this.#nodes.push(new Set());
    return id;
  }

  addEdge(nodeA, nodeB) {
    this.#nodes[nodeA].add(nodeB);
    this.#nodes[nodeB].add(nodeA);
  }
  neighbors(node) {
    return this.#nodes[node];
  }
}
```

When building up a graph, you call `addNode` to define a new node, and `addEdge` to connect two nodes together. Code working with the graph can use `neighbors`, which returns a set of connected node IDs, to read information about edges.

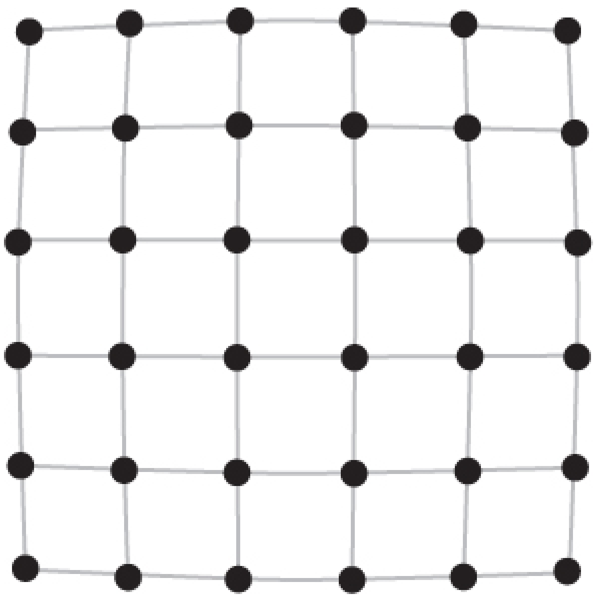
To represent a graph layout, we'll uses the familiar `Vec` class from previous chapters. A layout of a graph is an array of vectors with length `graph.size`, holding a position for each node.

```
function randomLayout(graph) {
  let layout = [];
  for (let i = 0; i < graph.size; i++) {
    layout.push(new Vec(Math.random() * 1000,
                        Math.random() * 1000));
  }
  return layout;
}
```

The `gridGraph` function builds a graph that is just a square grid of nodes, which is a useful test case for a graph layout program. It creates `size * size` nodes and connects each node with the node above or to the left of it (if there is such a node).

```
function gridGraph(size) {
  let grid = new Graph();
  for (let y = 0; y < size; y++) {
    for (let x = 0; x < size; x++) {
      let id = grid.addNode();
      if (x > 0) grid.addEdge(id, id - 1);
      if (y > 0) grid.addEdge(id, id - size);
    }
  }
  return grid;
}
```

This is what a layout for a grid might look like:



To allow us to inspect the layouts produced by our code, I've defined a `drawGraph` function that draws the graph onto a canvas. This function is defined in the code at *eloquentjavascript.net/code/draw_layout.js* and is available in the online sandbox.

Force-Directed Layout

We'll move nodes one at a time, computing the forces that act on the current node and immediately moving that node in the direction of the sum of these forces.

The force that an (idealized) spring applies can be approximated with Hooke's law, which says that this force is proportional to the difference between the spring's resting length and its current length. The binding `spring Length` defines the resting length of our edge springs. The rigidity of the springs is defined by `springStrength`, which we'll multiply by the length difference to determine the resulting force.

To model the repulsion between nodes, we use another physical formula, Coulomb's law, which says that the repulsion between two electrically charged particles is inversely proportional to the square of the distance between them. When two nodes are almost on top of each other, the squared distance is tiny and the resulting force is gigantic. As the nodes move farther apart, the squared distance grows rapidly so that the repelling force quickly weakens.

We'll multiply by an experimentally determined constant, `repulsion Strength`, which controls the strength with which nodes repel each other.

This function computes the strength of the force that acts on a node because of the presence of another node at a given distance. It always includes the repulsion force, and combines that with the force of the spring when the nodes are connected.

```
const springLength = 20;
const springStrength = 0.1;
const repulsionStrength = 1500;

function forceSize(distance, connected) {
  let repulse = -repulsionStrength / (distance * distar
  let spring = 0;
  if (connected) {
    spring = (distance - springLength) * springStrengtł
  }
  return spring + repulse;
}
```

The way a node moves is determined by combining the forces exercised on it by all the other nodes. For each pair of nodes, our function needs to know the distance between them. We can compute that by subtracting the positions of the nodes and taking the length of the resulting vector. When the distance is less than one, we set it to one to prevent dividing by zero or by very small numbers, as doing so would produce `NaN` values or forces so gigantic they catapult nodes into outer space.

Using this distance and the presence or absence of a connecting edge between the nodes, we can compute the magnitude of the force that acts between them. To go from this magnitude to a force vector, we can multiply the magnitude by a normalized version of the `apart` vector. *Normalizing* a vector means creating a vector with the same direction but with a length of one. We can do that by dividing the vector by its own length. Adding that to the node’s position moves it in the direction of the force.

```
function forceDirected_simple(layout, graph) {
  for (let a = 0; a < graph.size; a++) {
    for (let b = 0; b < graph.size; b++) {
      if (a == b) continue;
      let apart = layout[b].minus(layout[a]);
```

```
        let distance = Math.max(1, apart.length);
        let connected = graph.neighbors(a).has(b);
        let size = forceSize(distance, connected);
        let force = apart.times(1 / distance).times(size)
        layout[a] = layout[a].plus(force);
      }
    }
  }
```

We will use the following function to test a given implementation of our graph layout system. It start with a random layout and runs the model for three seconds. When finished, it logs the number of iterations it managed per second. To give us something to look at while the code runs, it draws the current layout of the graph after every 100 iterations.

```
function pause() {
  return new Promise(done => setTimeout(done, 0))
}

async function runLayout(implementation, graph) {
  let time = 0, iterations = 0;
  let layout = randomLayout(graph);
  while (time < 3000) {
    let start = Date.now();
    for (let i = 0; i < 100; i++) {
      implementation(layout, graph);
      iterations++;
    }
    time += Date.now() - start;
    drawGraph(graph, layout);
    await pause();
  }
  let perSecond = Math.round(iterations / (time / 1000)
  console.log(`${perSecond} iterations per second`);
}
```

To give the browser the opportunity to actually display the graph, the function briefly gives control back to the event loop every time it draws the graph. The function is asynchronous so that it can `await` the timeout.

We can run this first implementation and see how much time it takes.

```
runLayout(forceDirected_simple, gridGraph(12));
```

On my machine, this version manages about 1,600 iterations per second. That's quite a lot already. But let's see whether we can do better.

Avoiding Work

The fastest way to do something is to avoid doing it—or at least part of it—at all. By thinking about what the code is doing, you can often spot unnecessary redundance or things that can be done in a faster way.

In the case of our example project, there is such an opportunity for doing less work. Every pair of nodes has the forces between them computed twice, once when moving the first node and once when moving the second. Since the force that node *X* exerts on node *Y* is exactly the opposite of the force *Y* exerts on *X*, we do not need to compute these forces twice.

The next version of the function changes the inner loop to go over only the nodes that come after the current one so that each pair of nodes is looked at exactly once. After computing a force between a pair of nodes, the function updates the position of both.

```
function forceDirected_noRepeat(layout, graph) {
  for (let a = 0; a < graph.size; a++) {
    for (let b = a + 1; b < graph.size; b++) {
      let apart = layout[b].minus(layout[a]);
      let distance = Math.max(1, apart.length);
      let connected = graph.neighbors(a).has(b);
      let size = forceSize(distance, connected);
      let force = apart.times(1 / distance).times(size)
      layout[a] = layout[a].plus(force);
      layout[b] = layout[b].minus(force);
    }
  }
}
```

Apart from the loop structure and the fact that it adjusts both nodes, this version is identical to the previous one. Measuring this code shows a significant speed boost—about 45 percent faster on Chrome and 55 percent faster on Firefox.

Different JavaScript engines work differently and may run programs at different speeds. So a change that makes code run faster in one engine may not help (or may even hurt) in a different engine—or even a different version of the same engine. This is annoying, but not unexpected given the complexity of these systems.

If we take another good look at what our program is doing, possibly by calling `console.log` to output `size`, it becomes clear that the forces generated between most pairs of nodes are so tiny that they aren’t really impacting the layout at all. Specifically, when nodes are not connected and are far away from each other, the forces between them don’t amount to much. Yet we still compute vectors for them and move the nodes a tiny bit. What if we just didn’t?

This next version defines a distance above which pairs of (unconnected) nodes will no longer compute and apply forces. With that distance set to 175, forces below 0.05 are ignored.

```
const skipDistance = 175;

function forceDirected_skip(layout, graph) {
  for (let a = 0; a < graph.size; a++) {
    for (let b = a + 1; b < graph.size; b++) {
      let apart = layout[b].minus(layout[a]);
      let distance = Math.max(1, apart.length);
      let connected = graph.neighbors(a).has(b);
      if (distance > skipDistance && !connected) contir
      let size = forceSize(distance, connected);
      let force = apart.times(1 / distance).times(size)
      layout[a] = layout[a].plus(force);
      layout[b] = layout[b].minus(force);
    }
  }
}
```

This yields another 75 percent improvement in speed, with no discernable degradation of the layout. We cut a corner and got away with it.

Profiling

We were able to speed up the program quite a bit just by reasoning about it. But when it comes *micro-optimization*—the process of doing things slightly differently to make them faster—it is usually hard to predict which changes will help and which won't. There we can no longer rely on reasoning—we have to *observe*.

Our `runLayout` function measures the time the program currently takes. That's a good start. To improve something, you must measure it. Without measuring, you have no way of knowing whether your changes are having the effect you intended.

The developer tools in modern browsers provide an even better way to measure the speed of your program. This tool is called a *profiler*. It will, while a program is running, gather information about the time used by the various parts of the program.

If your browser has a profiler, it will be available from the developer tool interface, probably on a tab called Performance. The profiler in Chrome spits out the following table when I have it record 3,000 iterations of `forceDirected_skip`:

```
Activity               Self Time         Total Time
forceDirected_skip     74.0ms  82.4%     769.5ms  94.1%
Minor GC               48.2ms   5.9%      48.2ms   5.9%
Vec                    44.8ms   5.5%      46.9ms   5.7%
plus                    4.6ms   0.6%       5.5ms   0.7%
Optimize Code           0.1ms   0.0%       0.1ms   0.0%
```

This lists the functions (or other tasks) that took a serious amount of time. For each function, it reports the time spent executing the function, both in milliseconds and as a percentage of the total time taken. The first column shows only the time that control was actually in the function, whereas the second column includes time spent in functions called by this function.

As far as profiles go, this is a very simple one, since the program doesn't *have* a lot of functions. For more complicated programs, the lists will be longer.

Because the functions that take the most time are shown at the top, it is still usually easy to find the interesting information.

From this table, we can tell that most of the time is spent in the physics simulation function. That wasn't unexpected. But on the second row, we see "Minor GC." *GC* stands for "garbage collection"—the process of freeing up memory space taken up by values that the program no longer uses. The third row, which takes a similar amount of time, measures the vector constructor. These suggest the program is spending quite a lot of time creating and cleaning up `Vec` objects.

Imagine memory, again, as a long, long row of bits. When the program starts, it might receive an empty piece of memory and just start putting the objects it creates in there, one after the other. But at some point, the space is full, and some of the objects in it are no longer used. The JavaScript engine has to figure out which objects are used and which are not so that it can reuse the unused pieces of memory.

Each iteration of our loop creates five objects. The speed at which the engine is creating and reclaiming all these objects is already pretty impressive. Because many JavaScript programs create lots of objects, and managing memory for those objects can potentially slow the program down, a lot of effort has been spent on making this fast.

But as efficient as it is, it's still work that has to happen. Let's try a version of the code that doesn't create new vectors.

```
function forceDirected_noVector(layout, graph) {
  for (let a = 0; a < graph.size; a++) {
    let posA = layout[a];
    for (let b = a + 1; b < graph.size; b++) {
      let posB = layout[b];
      let apartX = posB.x - posA.x
      let apartY = posB.y - posA.y;
      let distance = Math.sqrt(apartX * apartX +
                               apartY * apartY);
      let connected = graph.neighbors(a).has(b);
      if (distance > skipDistance && !connected) contir
      let size = forceSize(distance, connected);
      let forceX = (apartX / distance) * size;
      let forceY = (apartY / distance) * size;
```

```
        posA.x += forceX;
        posA.y += forceY;
        posB.x -= forceX;
        posB.y -= forceY;
      }
    }
  }
```

The new code is wordier and more repetitive, but if I measure it, the improvement is large enough to consider doing this kind of manual object flattening in performance-sensitive code. On both Firefox and Chrome, the new version is about 50 percent faster than the previous one.

Taking all these steps together, we've made the program about four times faster than the initial version. That's quite an improvement. But remember that doing this work is useful only for code that actually takes a lot of time. Trying to optimize everything right away will only slow you down and leave you with a lot of needlessly overcomplicated code.

Function Inlining

Some vector methods (such as `times`) don't show up in the profile we saw, even though they were being used heavily. This is because the compiler *inlined* them. Rather than having the code in the inner function call an actual method to multiply vectors, the vector-multiplication code is put directly inside the function, and no actual method calls happen in the compiled code.

There are various ways in which inlining helps make code fast. Functions and methods are, at the machine level, called using a protocol that requires putting the arguments and the return address (the place where execution must continue when the function returns) somewhere the function can find them. The way a function call gives control to another part of the program also often requires saving some of the processor's state so that the called function can use the processor without interfering with data that the caller still needs. All of this becomes unnecessary when a function is inlined.

Furthermore, a good compiler will do its best to find ways to simplify the code it generates. If functions are treated as black boxes that might do anything, the compiler does not have a lot to work with. On the other hand, if it can see and include the function body in its analysis, it might find additional opportunities to optimize the code.

For example, a JavaScript engine could avoid creating some of the vector objects in our code altogether. In an expression like the following one, if we can see through the methods, it is clear that the resulting vector's coordinates are the result of adding the coordinates of `force` to the product of `normalized` and the `forceSize` binding. Thus, there is no need to create the intermediate object produced by the `times` method.

```
pos.plus(normalized.times(forceSize))
```

But JavaScript allows us to replace methods at any time by manipulating prototype objects. How does the compiler figure out which function this `times` method actually is? And what if someone changes the value stored in `Vec.prototype.times` later? The next time code that has inlined that function runs, it might continue to use the old definition, violating the programmer's assumptions about the way their program behaves.

This is where the interleaving of execution and compilation starts to pay off. When a hot function is compiled, it has already run a number of times. If, during those runs, it always called the same function, it is reasonable to try inlining that function. The code is optimistically compiled with the assumption that, in the future, the same function is going to be called here.

To handle the pessimistic case, where another function ends up being called, the compiler inserts a test that compares the called function to the one that was inlined. If the two do not match, the optimistically compiled code is wrong, and the JavaScript engine must *deoptimize*, meaning it falls back to a less optimized version of the code. At some later point, it may try to optimize again in a different way based on what it knows now.

Dynamic Types

JavaScript expressions like `graph.size`, which fetch a property from an object, are far from trivial to compile. In many languages, *bindings* have a type, and thus, when you perform an operation on the value they hold, the compiler already knows what kind of operation you need. In JavaScript, only *values* have types, and a binding can end up holding values of different types.

This means that initially the compiler knows little about the property the code might be trying to access and has to produce code that handles all possible types. If `graph` holds an undefined value, the code must throw an error. If it holds a string, it must look up `size` in `String.prototype`. If it holds an object, the way the `size` property is extracted from it depends on the shape of the object. And so on.

Fortunately, though JavaScript does not require it, bindings in most programs *do* have a single type. And if the compiler knows this type, it can use that information to produce efficient code. If `graph` has always been an instance of `Graph` so far, the optimizing compiler can create code that inlines the `size` getter from that class.

Again, events observed in the past don’t provide any guarantees about events that will occur in the future. Some piece of code that hasn’t run yet might still pass another type of value to our function—another kind of graph object, for example, whose `size` property works differently.

This means the compiled code still has to *check* whether its assumptions hold and take an appropriate action if they don’t. An engine could deoptimize entirely, falling back to the unoptimized version of the function. Or it could compile a new version of the function that also handles the newly observed type.

You can observe the slowdown caused by the failure to predict object types by intentionally messing up the uniformity of the input objects. For example, we could replace `randomLayout` with a version that adds a property with a random name to every vector.

```
function randomLayout(graph) {
  let layout = [];
  for (let i = 0; i < graph.size; i++) {
    let vector = new Vec(Math.random() * 1000,
                         Math.random() * 1000);
    vector[`p${Math.floor(Math.random() * 999)}`] = tru
    layout.push(vector);
  }
  return layout;
}

runLayout(forceDirected_noVector, gridGraph(12));
```

If we run our fast simulation code on the resulting graph, it becomes about three times as slow on Firefox and five times as slow on Chrome. Now that object types vary, the code that interacts with vectors has to look up the properties without prior knowledge about the shape of the object, which is a lot more expensive to do.

Interestingly, after running this code, `forceDirected_noVector` has become slow even when run on a regular, nonmangled layout vector. The messy types have "poisoned" the compiled code, at least for a while—at some point, browsers tend to throw away compiled code and recompile it from scratch, removing this effect.

A similar technique is used for things other than property access. The `+` operator, for example, means different things depending on what kind of values it is applied to. Instead of always running the full code that handles all these meanings, a smart JavaScript compiler will use previous observations to build up some expectation of the type that the operator is probably being applied to. If it is applied only to numbers, a much simpler piece of machine code can be generated to handle it. But again, such assumptions must be checked every time the function runs.

The lesson here is that if a piece of code needs to be fast, you can help by feeding it consistent types. JavaScript engines can handle cases where a few different types occur relatively well—they will generate code that handles all these types and deoptimizes only when a new type is seen. But even there, the resulting code is slower than what you would get for a single type.

Summary

Thanks to the enormous amount of money being poured into the web, as well as the rivalry between the different browsers, JavaScript compilers are good at what they do: making code run fast. But sometimes you have to help them a little and rewrite your inner loops to avoid more expensive JavaScript features. Creating fewer objects (and arrays and strings) often helps.

Before you start mangling your code to be faster, think about ways to make it do less work. The biggest opportunities for optimization are often found in that direction.

JavaScript engines compile hot code multiple times and will use information gathered during previous execution to compile more efficient code. Giving your bindings a consistent type helps make this type of optimization possible.

Exercises

Prime Numbers

Write a generator (see Chapter 11) `primes` that produces an endless stream of *prime numbers*. Prime numbers are whole numbers greater than 1 that cannot be divided by any whole number less than them and greater than 1. For example, the first five primes are 2, 3, 5, 7, and 11. Don't worry about speed (yet).

Set up a function `measurePrimes` that uses `Date.now()` to measure the time it takes your `primes` function to find the first ten thousand prime numbers.

Faster Prime Numbers

Now that you have a measured test case, find a way to make your `primes` function faster. Think about ways to reduce the number of remainder checks you have to perform.