JULIAN SHUN: Hi, good afternoon, everyone. So today, we're going to be talking about graph optimizations. And as a reminder, on Thursday, we're going to have a guest lecture by Professor Johnson of the MIT Math Department. And he'll be talking about performance of high-level languages. So please be sure to attend the guest lecture on Thursday.

So here's an outline of what I'm going to be talking about today. So we're first going to remind ourselves what a graph is. And then we're going to talk about various ways to represent a graph in memory. And then we'll talk about how to implement an efficient breadth-first search algorithm, both serially and also in parallel. And then I'll talk about how to use graph compression and graph reordering to improve the locality of graph algorithms.

So first of all, what is a graph? So a graph contains vertices and edges, where vertices represent certain objects of interest, and edges between objects model relationships between the two objects. For example, you can have a social network, where the people are represented as vertices and edges between people mean that they're friends with each other.

The edges in this graph don't have to be bi-directional. So you could have a one-way relationship. For example, if you're looking at the Twitter network, Alice could follow Bob, but Bob doesn't necessarily have to follow Alice back.

The graph also doesn't have to be connected. So here, this graph here is connected. But, for example, there could be some people who don't like to talk to other people. And then they're just off in their own component.

You can also use graphs to model protein networks, where the vertices are proteins, and edges between vertices means that there's some sort of interaction between the proteins. So this is useful in computational biology. As I said, edges can be directed, so their relationship can go one way or both ways.

In this graph here, we have some directed edges and then also some edges that are directed

in both directions. So here, John follows Alice. Alice follows Peter. And then Alice follows Bob, and Bob also follows Alice.

If you use a graph to represent the world wide web, then the vertices would be websites, and then the edges would denote that there is a hyperlink from one website to another. And again, the edges here don't have to be bi-directional because website A could have a link to website B. But website B doesn't necessarily have to have a link back.

Edges can also be weighted. So you can have a weight on the edge that denotes the strength of the relationship or some sort of distance measure corresponding to that relationship. So here, I have an example where I am using a graph to represent cities. And the edges between cities have a weight that corresponds to the distance between the two cities. And if I want to find the quickest way to get from city A to city B, then I would be interested in finding the shortest path from A to B in this graph here.

Here's another example, where the edge weights now are the costs of a direct flight from city A to city B. And here the edges are directed. So, for example, this says that there's a flight from San Francisco to LA for $45. And if I want to find the cheapest way to get from one city to another city, then, again, I would try to find the shortest path in this graph from city A to city B.

Vertices and edges can also have metadata on them, and they can also have types. So, for example, here's the Google Knowledge Graph, which represents all the knowledge on the internet that Google knows about. And here, the nodes have metadata on them. So, for example, the node corresponding to da Vinci is labeled with his date of birth and date of death.

And the vertices also have a color corresponding to the type of knowledge that they refer to. So you can see that some of these nodes are blue, some of them are red, some of them are green, and some of them have other things on them. So in general, graphs can have types and metadata on both the vertices as well as the edges.

Let's look at some more applications of graphs. So graphs are very useful for implementing queries on social networks. So here are some examples of queries that you might want to ask on a social network.

So, for example, you might be interested in finding all of your friends who went to the same high school as you on Facebook. So that can be implemented using a graph algorithm. You might also be interested in finding all of the common friends you have with somebody else--

again, a graph algorithm. And a social network service might run a graph algorithm to recommend people that you might know and want to become friends with. And they might use a graph algorithm to recommend certain products that you might be interested in.

So these are all examples of social network queries. And there are many other queries that you might be interested in running on a social network. And many of them can be implemented using graph algorithms.

Another important application is clustering. So here, the goal is to find groups of vertices in a graph that are well-connected internally and poorly-connected externally. So in this image here, each blob of vertices of the same color corresponds to a cluster. And you can see that inside a cluster, there are a lot of edges going among the vertices. And between clusters, there are relatively fewer edges.

And some applications of clustering include community detection and social networks. So here, you might be interested in finding groups of people with similar interests or hobbies. You can also use clustering to detect fraudulent websites on the internet.

You can use it for clustering documents. So you would cluster documents that have similar text together. And clustering is often used for unsupervised learning and machine learning applications.

Another application is connectomics. So connectomics is the study of the structure, the network structure of the brain. And here, the vertices correspond to neurons. And edges between two vertices means that there's some sort of interaction between the two neurons.

And recently, there's been a lot of work on trying to do high-performance connectomics. And some of this work has been going on here at MIT by Professor Charles Leiserson and Professor Nir Shavit's research group. So recently, this has been a very hot area.

Graphs are also used in computer vision-- for example, in image segmentation. So here, you want to segment your image into the distinct objects that appear in the image. And you can construct a graph by representing the pixels as vertices. And then you would place an edge between every pair of neighboring pixels with a weight that corresponds to their similarity. And then you would run some sort of minimum cost cut algorithm to partition your graph into the different objects that appear in the image.

So there are many other applications. And I'm not going to have time to go through all of them

today. But here's just a flavor of some of the applications of graphs. So any questions so far?

OK, so next, let's look at how we can represent a graph in memory. So for the rest of this lecture, I'm going to assume that my vertices are labeled in the range from 0 to n minus 1. So they have an integer in this range.

Sometimes, your graph might be given to you where the vertices are already labeled in this range, sometimes, not. But you can always get these labels by mapping each of the identifiers to a unique integer in this range. So for the rest of the lecture, I'm just going to assume that we have these labels from 0 to n minus 1 for the vertices.

One way to represent a graph is to use an adjacency matrix. So this is going to be n by n matrix. And there's a 1 bit in i-th row in j-th column if there's an edge that goes from vertex I to vertex J, and 0 otherwise. Another way to represent a graph is the edgeless representation, where we just store a list of the edges that appear in the graph. So we have one pair for each edge, where the pair contains the two coordinates of that edge.

So what is the space requirement for each of these two representations in terms of the number of edges m and the number of vertices n in the graph? So it should be pretty easy. Yes.

AUDIENCE: n squared for the [INAUDIBLE] and m for the [INAUDIBLE].

JULIAN SHUN: Yes, so the space for the adjacency matrix is order n squared because you have n squared cells in this matrix. And you have 1 bit for each of the cells. For the edge list, it's going to be order m because you have m edges. And for each edge, you're storing a constant amount of data in the edge list.

So here's another way to represent a graph. This is known as the adjacency list format. And idea here is that we're going to have an array of pointers, 1 per vertex. And each pointer points to a linked list storing the edges for that vertex. And the linked list is unordered in this example. So what's the space requirement of this representation?

AUDIENCE: It's n plus m.

JULIAN SHUN: Yeah, so it's going to be order n plus m. And this is because we have n pointers. And the number of entries across all of the linked lists is just equal to the number of edges in the graph, which is m.

What's one potential issue with this sort of representation if you think in terms of cache performance? Does anyone see a potential performance issue here? Yeah.

AUDIENCE: So it could be [INAUDIBLE].

JULIAN SHUN: Right. So the issue here is that if you're trying to loop over all of the neighbors of a vertex, you're going to have to dereference the pointer in every linked list node. Because these are not contiguous in memory. And every time you dereference linked lists node, that's going to be a random access into memory. So that can be bad for cache performance.

One way you can improve cache performance is instead of using linked lists for each of these neighbor lists, you can use an array. So now you can store the neighbors just in this array, and they'll be contiguous in memory. One drawback of this approach is that it becomes more expensive if you're trying to update the graph. And we'll talk more about that later. So any questions so far?

So what's another way to represent the graph that we've seen in a previous lecture? What's a more compressed or compact way to represent a graph, especially a sparse graph? So does anybody remember the compressed sparse row format?

So we looked at this in one of the early lectures. And in that lecture, we used it to store a sparse matrix. But you can also use it to store a sparse graph.

And as a reminder, we have two arrays in the compressed sparse row, or CSR format. We have the Offsets array and the Edges array. The Offsets array stores an offset for each vertex into the Edges array, telling us where the edges for that particular vertex begins in the Edges array.

So Offsets of i stores the offset of where vertex i's edges start in the Edges array. So in this example, vertex 0 has an offset of 0. So its edges start at position 0 in the Edges array. Vertex 1 has an offset of 4, so it starts at index 4 in this Offsets array.

So with this representation, how can we get the degree of a vertex? So we're not storing the degree explicitly here. Can we get the degree efficiently? Yes.

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so you can get the degree of a vertex just by looking at the difference between the next

offset and its own offset. So for vertex 0, you can see that its degree is 4 because vertex 1's offset is 4, and vertex 0's offset is 0. And similarly you can do that for all of the other vertices. So what's the space usage of this representation?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Sorry, can you repeat?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so again, it's going to be order m plus n because you need order n space for the Offsets array and order m space for the Edges array. You can also store values or weights on their edges. One way to do this is to create an additional array of size m. And then for edge i, you just store the weight or the value in the i-th index of this additional array that you created.

If you're always accessing the weight when you access an edge, then it's actually better for a cache locality to interleave the weights with the edge targets. So instead of creating two arrays of size m, you have one array of size 2m. And every other entry is the weight.

And this improves cache locality because every time you access an edge, its weight is going to be right next to it in memory. And it's going to likely be on the same cache line. So that's one way to improve cache locality. Any questions so far?

So let's look at some of the trade-offs in these different graph representations that we've looked at so far. So here, I'm listing the storage costs for each of these representations which we already discussed. This is also the cost for just scanning the whole graph in one of these representations. What's the cost of adding an edge in each of these representations? So for adjacency matrix, what's the cost of adding an edge?

AUDIENCE: Order 1.

JULIAN SHUN: So for adjacency matrix, it's just order 1 to add an edge. Because you have random access into this matrix, so you just have to access to i, j-th entry and flip the bit from 0 to 1. What about for the edge list? So assuming that the edge list is unordered, so you don't have to keep the list in any sorted order. Yeah.

AUDIENCE: I guess it's O of 1.

JULIAN SHUN: Yeah, so again, it's just O of 1 because you can just add it to the end of the edge list. So that's a constant time. What about for the adjacency list?

So actually, this depends on whether we're using linked lists or arrays for the neighbor lists of the vertices. If we're using a linked list, adding an edge just takes constant time because we can just put it at the beginning of the linked list. If we're using an array, then we actually need to create a new array to make space for this edge that we add.

And that's going to cost us a degree of v work to do that because we have to copy all the existing edges over to this new array and then add this new edge to the end of that array. Of course, you could amortize this cost across multiple updates. So if you run out of memory, you can double the size of your array so you don't have to create these new arrays too often. But the cost for any individual addition is still relatively expensive compared to, say, an edge list or adjacency matrix.

And then finally, for the compressed sparse row format, if you add an edge, in the worst case, it's going to cost us order m plus n work. Because we're going to have to reconstruct the entire Offsets array and the entire Edges array in the worst case. Because we have to put something in and then shift-- in the Edges array, you have to put something in and shift all of the values to the right of that over by one location. And then for the Offsets array, we have to modify the offset for the particular vertex we're adding an edge to and then the offsets for all of the vertices after that.

So the compressed sparse row representation is not particularly friendly to edge updates. What about for deleting an edge from some vertex v? So for adjacency matrix, again, it's going to be constant time because you just randomly access the correct entry and flip the bit from 1 to 0. What about for an edge list?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so for an edge list, in the worst case, it's going to cost us order m work because the edges are not in any sorted order. So we have to scan through the whole thing in the worst case to find the edge that we're trying to delete. For adjacency list, it's going to take order degree of v work because the neighbors are not sorted. So we have to scan through the whole thing to find this edge that we're trying to delete. And then finally, for a compressed sparse row, it's going to be order m plus n because we're going to have to reconstruct the whole thing in the worst case.

What about finding all of the neighbors of a particular vertex v? What's the cost of doing this in the adjacency matrix?

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yes, so it's going to cost us order n work to find all the neighbors of a particular vertex because we just scan the correct row in this matrix, the row corresponding to vertex v. For the edge list, we're going to have to scan the entire edge list because it's not sorted. So in the worst case, that's going to be order m.

For adjacency list, that's going to take order degree of v because we can just find a pointer to the linked list for that vertex in constant time. And then we just traverse over the linked list. And that takes order degree of v time. And then finally, for compressed sparse row format, it's also order degree of v because we have constant time access into the appropriate location in the Edges array. And then we can just read off the edges, which are consecutive in memory.

So what about finding if a vertex w is a neighbor of v? So I'll just give you the answer. So for the adjacency matrix, it's going to take constant time because again, we just have to check the v-th row in the w-th column and check if the bit is set there. For edge list, we have to traverse the entire list to see if the edge is there. And then for adjacency list and compressed sparse row, it's going to be order degree of v because we just have to scan the neighbor list for that vertex.

So these are some graph representations. But there are actually many other graph representations, including variance of the ones that I've talked about here. So, for example, for the adjacency, I said you can either use a linked list or an array to store the neighbor list. But you can actually use a hybrid approach, where you store the linked list, but each linked list node actually stores more than one vertex. So you can store maybe 16 vertices in each linked list node. And that gives us better cache locality.

So for the rest of this lecture, I'm going to talk about algorithms that are best implemented using the compressed sparse row format. And this is because we're going to be dealing with sparse graphs. We're going to be looking at static algorithms, where we don't have to update the graph. If we do have to update the graph, then CSR isn't a good choice. But we're just going to be looking at static algorithms today.

And then for all the algorithms that we'll be looking at, we're going to need to scan over all the neighbors of a vertex that we visit. And CSR is very good for that because all of the neighbors for a particular vertex are stored contiguously in memory. So any questions so far?

OK, I do want to talk about some properties of real-world graphs. So first, we're seeing graphs that are quite large today. But actually, they're not too large. So here are the sizes of some of the real-world graphs out there.

So there is a Twitter network. That's actually a snapshot of the Twitter network from a couple of years ago. It has 41 million vertices and 1.5 billion edges. And you can store this graph in about 6.3 gigabytes of memory. So you can probably store it in the main memory of your laptop.

The largest publicly available graph out there now is this Common Crawl web graph. It has 3.5 billion vertices and 128 billion edges. So storing this graph requires a little over 1/2 terabyte of memory. It is quite a bit of memory. But it's actually not too big because there are machines out there with main memory sizes in the order of terabytes of memory nowadays.

So, for example, you can rent 2-terabyte or 4-terabyte memory instance on AWS, which you're using for your homework assignments. See if you have any leftover credits at the end of the semester, and you want to play around on this graph, you can rent one of these terabyte machines. Just remember to turn it off when you're done because it's kind of expensive.

Another property of real-world graphs is that they're quite sparse. So m tends to be much less than n squared. So most of the possible edges are not actually there.

And finally, the degree distributions of the vertices can be highly skewed in many real-world graphs. So here I'm plotting the degree on the x-axis and the number of vertices with that particular degree on the y-axis. And we can see that it's highly skewed. And, for example, in a social network, most of the people would be on the left-hand side, so their degree is not that high. And then we have some very popular people on the right-hand side, where their degree is very high, but we don't have too many of those.

So this is what's known as a power law degree distribution. And there have been various studies that have shown that many real-world graphs have approximately a power law degree distribution. And mathematically, this means that the number of vertices with degree d is proportional to d to the negative p. So negative p is the exponent.

And for many graphs, the value of p lies between 2 and 3. And this power law degree distribution does have implications when we're trying to implement parallel algorithms to process these graphs. Because with graphs that have a skewed degree distribution, you could run into load and balance issues. If you just parallelize across the vertices, the number of edges they have can vary significantly. Any questions?

OK, so now let's talk about how we can implement a graph algorithm. And I'm going to talk about the breadth-first search algorithm. So how many of you have seen breadth-first search before? OK, so about half of you. I did talk about breadth-first search in a previous lecture, so I was hoping everybody would raise their hands.

OK, so as a reminder, in the BFS algorithm, we're given a source vertex s, and we want to visit the vertices in order of their distance from the source s. And there are many possible outputs that we might care about. One possible output is, we just want to report the vertices in the order that they were visited by the breadth-first search traversal.

So let's say we have this graph here. And our source vertex is D. So what's one possible order in which we can traverse these vertices? Now, I should specify that we should traverse this graph in a breadth-first search manner. So what's the first vertex we're going to explore?

AUDIENCE: D.

JULIAN SHUN: D. So we're first going to look at D because that's our source vertex. The second vertex, we can actually choose between B, C, and E because all we care about is that we're visiting these vertices in the order of their distance from the source. But these three vertices are all of the same distance. So let's just pick B, C, and then E.

And then finally, I'm going to visit vertex A, which has a distance of 2 from the source. So this is one possible solution. There are other possible solutions because we could have visited E before we visited B and so on. Another possible output that we might care about is we might want to report the distance from each vertex to the source vertex s. So in this example here are the distances. So D has a distance of 0; B,C, and E all have a distance of 1; and A has a distance of 2.

We might also want to generate a breadth-first search tree where each vertex in the tree has a parent which is a neighbor in the previous level of the breadth-first search. Or in other words, the parent should have a distance of 1 less than that vertex itself. So here's an example of a

breadth-first search tree. And we can see that each of the vertices has a parent whose breadth-first search distance is 1 less than itself.

So the algorithms that I'm going to be talking about today will generate the distances as well as the BFS tree. And BFS actually has many applications. So it's used as a subroutine in betweenness centrality, which is a very popular graph mining algorithm used to rank the importance of nodes in a network. And the importance of nodes here corresponds to how many shortest paths go through that node.

Other applications include eccentricity estimation, maximum flows. Some max flow algorithms use BFS as a subroutine. You can use BFS to crawl the web, do cycle detection, garbage collection, and so on.

So let's now look at a serial BFS algorithm. And here, I'm just going to show the pseudocode. So first, we're going to initialize the distances to all INFINITY. And we're going to initialize the parents to be NIL.

And then we're going to create queue data structure. We're going to set the distance of the root to be 0 because the root has a distance of 0 to itself. And then we're going to place the root onto this queue.

And then, while the queue is not empty, we're going to dequeue the first thing in the queue. We're going to look at all the neighbors of the current vertex that we dequeued. And for each neighbor, we're going to check if its distance is INFINITY.

If the distance is INFINITY, that means we haven't explored that neighbor yet. So we're going to go ahead and explore it. And we do so by setting its distance value to be the current vertex's distance plus 1. We're going to set the parent of that neighbor to be the current vertex. And then we'll place the neighbor onto the queue.

So it's some pretty simple algorithm. And we're just going to keep iterating in this while loop until there are no more vertices left in the queue. So what's the work of this algorithm in terms of n and m? So how much work are we doing per edge? Yes.

AUDIENCE: [INAUDIBLE]

JULIAN SHUN: Yeah, so assuming that the enqueue and dequeue operators are constant time, then we're doing constant amount of work per edge. So summed across all edges, that's going to be

order m. And then we're also doing a constant amount of work per vertex because we have to basically place it onto the queue and then take it off the queue, and then also initialize their value. So the overall work is going to be order m plus n.

OK, so let's now look at some actual code to implement the serial BFS algorithm using the compressed sparse row format. So first, I'm going to initialize two arrays-- parent and queue. And these are going to be integer arrays of size n.

I'm going to initialize all of the parent entries to be negative 1. I'm going to place a source vertex onto the queue. So it's going to appear at queue of 0, that's the beginning of the queue.

And then I'll set the parent of the source vertex to be the source itself. And then I also have two integers that point to the front and the back of the queue. So initially, the front of the queue is at position 0, and the back is at position 1.

And then while the queue is not empty-- and I can check that by checking if q_front is not equal to q_back-- then I'm going to dequeue the first vertex in my queue. I'm going to set current to be that vertex. And then I'll increment q_front. And then I'll compute the degree of that vertex, which I can do by looking at the difference between consecutive offsets. And I also assume that Offsets of n is equal to m, just to deal with the last vertex

And then I'm going to loop through all of the neighbors for the current vertex. And to access each neighbor, what I do is I go into the Edges array. And I know that my neighbors start at Offsets of current. And therefore, to get the i-th neighbor, I just do Offsets of current plus i. That's my index into the Edges array.

Now I'm going to check if my neighbor has been explored yet. And I can check that by checking if parent of neighbor is equal to negative 1. If it is, that means I haven't explored it yet. And then I'll set a parent of neighbor to be current. And then I'll place the neighbor onto the back of the queue and increment q_back.

And I'm just going to keep repeating this while loop until it becomes empty. And here, I'm only generating the parent pointers. But I could also generate the distances if I wanted to with just a slight modification of this code.

So any questions on how this code works? OK, so here's a question. What's the most expensive part of the code? Can you point to one particular line here that is the most expensive? Yes.

AUDIENCE: I'm going to guess the [INAUDIBLE] that's gonna be all over the place in terms of memory locations-- ngh equals Edges.

JULIAN SHUN: OK, so actually, it turns out that that's not the most expensive part of this code. But you're close. So anyone have any other ideas? Yes.

AUDIENCE: Is it looking up the parent array?

JULIAN SHUN: Yes, so it turns out that this line here, where we're accessing parent of neighbor, that turns out to be the most expensive. Because whenever we access this parent array, the neighbor can appear anywhere in memory. So that's going to be a random access. And if the parent array doesn't fit in our cache, then that's going to cost us a cache miss almost every time.

This Edges array is actually mostly accessed sequentially. Because for each vertex, all of its edges are stored contiguously in memory, we do have one random access into the Edges array per vertex because we have to look up the starting location for that vertex. But it's not 1 per edge, unlike this check of the parent array. That occurs for every edge. So does that make sense?

So let's do a back-of-the-envelope calculation to figure out how many cache misses we would incur, assuming that we started with a cold cache. And we also assume that n is much larger than the size of the cache, so we can't fit any of these arrays into cache. We'll assume that a cache line has 64 bytes, and integers are 4 bytes each.

So let's try to analyze this. So the initialization will cost us n/16 cache misses. And the reason here is that we're initializing this array sequentially. So we're accessing contiguous locations.

And this can take advantage of spatial locality. On each cache line, we can fit 16 of the integers. So overall, we're going to need n/16 cache misses just to initialize this array.

We also need n/16 cache misses across the entire algorithm to dequeue the vertex from the front of the queue. Because again, this is going to be a sequential access into this queue array. And across all vertices, that's going to be n/16 cache misses because we can fit 16 integers on a cache line.

To compute the degree here, that's going to take n cache misses overall. Because each of these accesses to Offsets array is going to be a random access. Because we have no idea

what the value of current here is. It could be anything.

So across the entire algorithm, we're going to need n cache misses to access this Offsets array. And then to access this Edges array, I claim that we're going to need at most 2n plus m/16 cache misses. So does anyone see where that bound comes from? So where does the m/16 come from? Yeah.

AUDIENCE: You have to access that at least once for an edge.

JULIAN SHUN: Right, so you have to pay m/16 because you're accessing every edge once. And you're accessing the Edges contiguously. So therefore, across all Edges, that's going to take m/16 cache misses.

But we also have to add 2n. Because whenever we access the Edges for a particular vertex, the first cache line might not only contain that vertex's edges. And similarly, the last cache line that we access might also not just contain that vertex's edges.

So therefore, we're going to waste the first cache line and the last cache line in the worst case for each vertex. And summed cross all vertices, that's going to be 2n. So this is the upper bound, 2n plus m/16.

Accessing this parent array, that's going to be a random access every time. So we're going to incur a cache miss in the worst case every time. So summed across all edge accesses, that's going to be m cache misses. And then finally, we're going to pay n/16 cache misses to enqueue the neighbor onto the queue because these are sequential accesses.

So in total, we're going to incur at most 51/16 n plus 17/16 16 m cache misses. And if m is greater than 3n, then the second term here is going to dominate. And m is usually greater than 3n in most real-world graphs. And the second term here is dominated by this random access into the parent array.

So let's see if we can optimize this code so that we get better cache performance. So let's say we could fit a bit vector of size n into cache. But we couldn't fit the entire parent array into cache. What can we do to reduce the number of cache misses? So does anyone have any ideas? Yeah.

AUDIENCE: Is bitvector to keep track of which vertices of other parents then [INAUDIBLE]?

JULIAN SHUN: Yeah, so that's exactly correct. So we're going to use a bit vector to store whether the vertex has been explored yet or not. So we only need 1 bit for that. We're not storing the parent ID in this bit vector. We're just storing a bit to say whether that vertex has been explored yet or not.

And then, before we check this parent array, we're going to first check the bit vector to see if that vertex has been explored yet. And if it has been explored yet, we don't even need to access this parent array. If it hasn't been explored, then we won't go ahead and access the parent entry of the neighbor. But we only have to do this one time for each vertex in the graph because we can only visit each vertex once. And therefore, we can reduce the number of cache misses from m down to n.

So overall, this might improve the number of cache misses. In fact, it does if the number of edges is large enough relative to the number of vertices. However, you do have to do a little bit more computation because you have to do bit vector manipulation to check this bit vector and then also to set the bit vector when you explore a neighbor. So here's the code using the bit vector optimization.

So here, I'm initializing this bit vector called visited. It's of size, approximately, n/32. And then I'm setting all of the bits to 0, except for the source vertex, where I'm going to set its bit to 1. And I'm doing this bit calculation here to figure out the bit for the source vertex.

And then now, when I'm trying to visit a neighbor, I'm first going to check if the neighbor is visited by checking this bit array. And I can do this using this computation here-- AND visited of neighbor over 32, by this mask-- 1 left shifted by neighbor mod 32. And if that's false, that means the neighbor hasn't been visited yet.

So I'll go inside this IF clause. And then I'll set the visited bit to be true using this statement here. And then I do the same operations as I did before.

It turns out that this version is faster for large enough values of m relative to n because you reduce the number of cache misses overall. You still have to do this extra computation here, this bit manipulation. But if m is large enough, then the reduction in number of cache misses outweighs the additional computation that you have to do. Any questions?

OK, so that was a serial implementation of breadth-first search. Now let's look at a parallel implementation. So I'm first going to do an animation of how a parallel breadth-first search algorithm would work.

The parallel reference search algorithm is going to operate on frontiers, where the initial frontier contains just a source vertex. And on every iteration, I'm going to explore all of the vertices on the frontier and then place any unexplored neighbors onto the next frontier. And then I move on to the next frontier.

So in the first iteration, I'm going to mark the source vertex as explored, set its distance to be 0, and then place the neighbors of that source vertex onto the next frontier. In the next iteration, I'm going to do the same thing, set these distances to 1. I also am going to generate a parent pointer for each of these vertices. And this parent should come from the previous frontier, and it should be a neighbor of the vertex.

And here, there's only one option, which is the source vertex. So I'll just pick that as the parent. And then I'm going to place the neighbors onto the next frontier again, mark those as explored, set their distances, and generate a parent pointer again.

And notice here, when I'm generating these parent pointers, there's actually more than one choice for some of these vertices. And this is because there are multiple vertices on the previous frontier. And some of them explored the same neighbor on the current frontier. So a parallel implementation has to be aware of this potential race. Here, I'm just picking an arbitrary parent.

So as we see here, you can process each of these frontiers in parallel. So you can parallelize over all of the vertices on the frontier as well as all of their outgoing edges. However, you do need to process one frontier before you move on to the next one in this BFS algorithm.

And a parallel implementation has to be aware of potential races. So as I said earlier, we could have multiple vertices on the frontier trying to visit the same neighbors. So somehow, that has to be resolved.

And also, the amount of work on each frontier is changing throughout the course of the algorithm. So you have to be careful with load balancing. Because you have to make sure that the amount of work each processor has to do is about the same. If you use Cilk to implement this, then load balancing doesn't really become a problem. So any questions on the BFS algorithm before I go over the code?

OK, so here's the actual code. And here I'm going to initialize these four arrays, so the parent array, which is the same as before. I'm going to have an array called frontier, which stores the

current frontier. And then I'm going to have an array called frontierNext, which is a temporary array that I use to store the next frontier of the BFS. And then also I have an array called degrees.

I'm going to initialize all of the parent entries to be negative 1. I do that using a cilk_for loop. I'm going to place the source vertex at the 0-th index of the frontier. I'll set the frontierSize to be 1. And then I set the parent of the source to be the source itself.

While the frontierSize is greater than 0, that means I still have more work to do. I'm going to first iterate over all of the vertices on my frontier in parallel using a cilk_for loop. And then I'll set the i-th entry of the degrees array to be the degree of the i-th vertex on the frontier. And I can do this just using the difference between consecutive offsets.

And then I'm going to perform a prefix sum on this degrees array. And we'll see in a minute why I'm doing this prefix sum. But first of all, does anybody recall what prefix sum is? So who knows what prefix sum is? Do you want to tell us what it is?

AUDIENCE: That's the sum array where index i is the sum of [INAUDIBLE].

JULIAN SHUN: Yeah, so prefix sum-- so here I'm going to demonstrate this with an example. So let's say this is our input array. The output of this array would store for each location the sum of everything before that location in the input array.

So here we see that the first position has a value of 0 because a sum of everything before it is 0. There's nothing before it in the input. The second position has a value of 2 because the sum of everything before it is just the first location. The third location has a value of 6 because the sum of everything before it is 2 plus 4, which is 6, and so on.

So I believe this was on one of your homework assignments. So hopefully, everyone knows what prefix sum is. And later on, we'll see how we use this to do the parallel breadth-first search.

OK, so I'm going to do a prefix sum on this degrees array. And then I'm going to loop over my frontier again in parallel. I'm going to let v be the i-th vertex on the frontier. Index is going to be equal to degrees of i. And then my degree is going to be Offsets of v plus 1 minus Offsets of v.

Now I'm going to loop through all v's neighbors. And here I just have a serial for loop. But you could actually parallelize this for loop. It turns out that if the number of iterations in the for loop

is small enough, there's additional overhead to making this parallel, so I just made it serial for now. But you could make it parallel.

To get the neighbor, I just index into this Edges array. I look at Offsets of v plus j. Then now I'm going to check if the neighbor has been explored yet.

And I can check if parent of neighbor is equal to negative 1. So that means it hasn't been explored yet, so I'm going to try to explore it. And I do so using a compare-and-swap.

I'm going to try to swap in the value of v with the original value of negative 1 in parent of neighbor. And the compare-and-swap is going to return true if it was successful and false otherwise. And if it returns true, that means this vertex becomes the parent of this neighbor. And then I'll place the neighbor on to frontierNext at this particular index-- index plus j. And otherwise, I'll set a negative 1 at that location.

OK, so let's see why I'm using index plus j here. So here's how frontierNext is organized. So each vertex on the frontier owns a subset of these locations in the frontierNext array. And these are all contiguous memory locations.

And it turns out that the starting location for each of these vertices in this frontierNext array is exactly the value in this prefix sum array up here. So vertex 1 has its first location at index 0. Vertex 2 has its first location at index 2. Vertex 3 has its first location at index 6, and so on.

So by using a prefix sum, I can guarantee that all of these vertices have a disjoint subarray in this frontierNext array. And then they can all write to this frontierNext array in parallel without any races. And index plus j just gives us the right location to write to in this array. So index is the starting location, and then j is for the j-th neighbor.

So here is one potential output after we write to this frontierNext array. So we have some non-negative values. And these are vertices that we explored in this iteration.

We also have some negative 1 values. And the negative 1 here means that either the vertex has already been explored in a previous iteration, or we tried to explore it in the current iteration, but somebody else got there before us. Because somebody else is doing the compare-and-swap at the same time, and they could have finished before we did, so we failed on the compare-and-swap.

So we don't actually want these negative 1 values, so we're going to filter them out. And we

can filter them out using a prefix sum again. And this is going to give us a new frontier.

And we'll set the frontierSize equal to the size of this new frontier. And then we repeat this while loop until there are no more vertices on the frontier. So any questions on this parallel BFS algorithm? Yeah.

AUDIENCE: Can you go over like the last [INAUDIBLE]?

JULIAN SHUN: Do you mean the filter out?

AUDIENCE: Yeah.

JULIAN SHUN: Yeah, so what you can do is, you can create another array, which stores a 1 in location i if that location is not a negative 1 and 0 if it is a negative 1. Then you do a prefix sum on that array, which gives us unique offsets into an output array. So then everybody just looks at the prefix sum array there. And then it writes to the output array. So it might be easier if I tried to draw this on the board.

OK, so let's say we have an array of size 5 here. So what I'm going to do is I'm going to generate another array which stores a 1 if the value in the corresponding location is not a negative 1 and 0 otherwise. And then I do a prefix sum on this array here. And this gives me 0, 1, 1, 2, and 2.

And now each of these values that are not negative 1, they can just look up the corresponding index in this output array. And this gives us a unique index into an output array. So this element will write to position 0, this element would write to position 1, and this element would write to position 2 in my final output. So this would be my final frontier. Does that make sense?

OK, so let's now analyze the working span of this parallel BFS algorithm. So a number of iterations required by the BFS algorithm is upper-bounded by the diameter D of the graph. And the diameter of a graph is just the maximum shortest path between any pair of vertices in the graph. And that's an upper bound on the number of iterations we need to do.

Each iteration is going to take a log m span for the clik_for loops, the prefix sum, and the filter. And this is also assuming that the inner loop is parallelized, the inner loop over the neighbors of a vertex. So to get the span, we just multiply these two terms. So we get theta of D times log m span.

What about the work? So to compute the work, we have to figure out how much work we're doing per vertex and per edge. So first, notice that the sum of the frontier sizes across entire algorithm is going to be n because each vertex can be on the frontier at most once. Also, each edge is going to be traversed exactly once. So that leads to m total edge visits.

On each iteration of the algorithm, we're doing a prefix sum. And the cost of this prefix sum is going to be proportional to the frontier size. So summed across all iterations, the cost of the prefix sum is going to be theta of n.

We also have to do this filter. But the work of the filter is proportional to the number of edges traversed in that iteration. And summed across all iterations, that's going to give theta of m total.

So overall, the work is going to be theta of n plus m for this parallel BFS algorithm. So this is a work-efficient algorithm. The work matches out the serial algorithm. Any questions on the analysis?

OK, so let's look at how this parallel BFS algorithm runs in practice. So here, I ran some experiments on a random graph with 10 million vertices and 100 million edges. And the edges were randomly generated.

And I made sure that each vertex had 10 edges. I ran experiments on a 40-core machine with 2-way hyperthreading. Does anyone know what hyperthreading is? Yeah, what is it?

AUDIENCE: It's when you have like one CPU core that can execute two instruction screens at the same time so it can [INAUDIBLE] high number latency.

JULIAN SHUN: Yeah, so that's a great answer. So hyperthreading is an Intel technology where for each physical core, the operating system actually sees it as two logical cores. They share many of the same resources, but they have their own registers. So if one of the logical cores stalls on a long latency operation, the other logical core can use the shared resources and hide some of the latency.

OK, so here I am plotting the speedup over the single-threaded time of the parallel algorithm versus the number of threads. So we see that on 40 threads, we get a speedup of about 22 or 23X. And when we turn on hyperthreading and use all 80 threads, the speedup is about 32 times on 40 cores.

And this is actually pretty good for a parallel graph algorithm. It's very hard to get very good speedups on these irregular graph algorithms. So 32X on 40 cores is pretty good.

I also compared this to the serial BFS algorithm because that's what we ultimately want to compare against. So we see that on 80 threads, the speedup over the serial BFS is about 21, 22X. And the serial BFS is 54% faster than the parallel BFS on one thread.

This is because it's doing less work than the parallel version. The parallel version has to do actual work with the prefix sum in the filter, whereas the serial version doesn't have to do that. But overall, the parallel implementation is still pretty good. OK, questions?

So a couple of lectures ago, we saw this slide here. So Charles told us never to write nondeterministic parallel programs because it's very hard to debug these programs and hard to reason about them. So is there nondeterminism in this BFS code that we looked at?

AUDIENCE: You have nondeterminism in the compare-and-swap.

JULIAN SHUN: Yeah, so there's nondeterminism in the compare-and-swap. So let's go back to the code. So this compare-and-swap here, there's a race there because we get multiple vertices trying to write to the parent entry of the neighbor at the same time. And the one that wins is nondeterministic. So the BFS tree that you get at the end is nondeterministic.

OK, so let's see how we can try to fix this nondeterminism. OK so, as we said, this is a line that causes the nondeterminism. It turns out that we can actually make the output BFS tree, be deterministic by going over the outgoing edges in each iteration in two phases.

So how this works is that in the first phase, the vertices on the frontier are not actually going to write to the parent array. Or they are going to write, but they're going to be using this writeMin operator. And the writeMin operator is an atomic operation that guarantees that we have concurrent writes to the same location.

The smallest value gets written there. So the value that gets written there is going to be deterministic. It's always going to be the smallest one that tries to write there.

Then in the second phase, each vertex is going to check for each neighbor whether a parent of neighbor is equal to v. If it is, that means it was the vertex that successfully wrote to parent of neighbor in the first phase. And therefore, it's going to be responsible for placing this neighbor onto the next frontier.

And we're also going to set parent of neighbor to be negative v. This is just a minor detail. And this is because when we're doing this writeMin operator, we could have a future iteration where a lower vertex tries to visit the same vertex that we already explored. But if we set this to a negative value, we're only going to be writing non-negative values to this location. So the writeMin on a neighbor that has already been explored would never succeed.

OK, so the final BFS tree that's generated by this code is always going to be the same every time you run it. I want to point out that this code is still notdeterministic with respect to the order in which individual memory locations get updated. So you still have a deterministic race here in the writeMin operator. But it's still better than a nondeterministic code in that you always get the same BFS tree.

So how do you actually implement the writeMin operation? So it turns out you can implement this using a loop with a compare-and-swap. So writeMin takes as input two arguments-- the memory address that we're trying to update and the new value that we want to write to that address.

We're first going to set oldval equal to the value at that memory address. And we're going to check if newval is less than oldval. If it is, then we're going to attempt to do a compare-and-swap at that location, writing newval into that address if its initial value was oldval.

And if that succeeds, then we return. Otherwise, we failed. And that means that somebody else came in the meantime and changed the value there. And therefore, we have to reread the old value at the memory address. And then we repeat.

And there are two ways that this writeMin operator could finish. One is if the compare-and-swap was successful. The other one is if newval is greater than or equal to oldval. In that case, we no longer have to try to write anymore because the value that's there is already smaller than what we're trying to write.

So I implemented an optimized version of this deterministic parallel BFS code and compared it to the nondeterministic version. And it turns out on 32 cores, it's only a little bit slower than the nondeterministic version. So it's about 5% to 20% slower on a range of different input graphs. So this is a pretty small price to pay for determinism. And you get many nice benefits, such as ease of debugging and ease of reasoning about the performance of your code. Any questions?

OK, so let me talk about another optimization for breadth-first search. And this is called the direction optimization. And the idea is motivated by how the sizes of the frontiers change in a typical BFS algorithm over time.

So here I'm plotting the frontier size on the y-axis in log scale. And the x-axis is the iteration number. And on the left, we have a random graph, on the right, we have a parallel graph.

And we see that the frontier size actually grows pretty rapidly, especially for the power law graph. And then it drops pretty rapidly. So this is true for many of the real-world graphs that we see because many of them look like power law graphs. And in the BFS algorithm, most of the work is done when the frontier is relatively large. So most of the work is going to be done in these middle iterations where the frontier is very large.

And it turns out that there are two ways to do breadth-first search. One way is the traditional way, which I'm going to refer to as the top-down method. And this is just what we did before. We look at the frontier vertices, and explore all of their outgoing neighbors, and mark any of the unexplored ones as explored, and place them on to the next frontier.

But there's actually another way to do breadth-first search. And this is known as the bottom-up method. And in the bottom-up method, I'm going to look at all of the vertices in the graph that haven't been explored yet, and I'm going to look at their incoming edges.

And if I find an incoming edge that's on the current frontier, I can just say that that incoming neighbor is my parent. And I don't even need to look at the rest of my incoming neighbors. So in this example here, vertices 9 through 12, when they loop through their incoming edges, they found incoming neighbor on the frontier, and they chose that neighbor as their parent. And they get marked as explored.

And we can actually save some edge traversals here because, for example, if you look at vertex 9, and you imagine the edges being traversed in a top-to-bottom manner, then vertex 9 is only going to look at its first incoming edge and find the incoming neighbors on the frontier. So it doesn't even need to inspect the rest of the incoming edges because all we care about finding is just one parent in the BFS tree. We don't need to find all of the possible parents.

In this example here, vertices 13 through 15 actually ended up wasting work because they looked at all of their incoming edges. And none of the incoming neighbors are on the frontier. So they don't actually find a neighbor.

So the bottom-up approach turns out to work pretty well when the frontier is large and many vertices have been already explored. Because in this case, you don't have to look at many vertices. And for the ones that you do look at, when you scan over their incoming edges, it's very likely that early on, you'll find a neighbor that is on the current frontier, and you can skip a bunch of edge traversals. And the top-down approach is better when the frontier is relatively small.

And in a paper by Scott Beamer in 2012, he actually studied the performance of these two approaches in BFS. And this plot here plots the running time versus the iteration number for a power law graph and compares the performance of the top-down and bottom-up approach. So we see that for the first two steps, the top-down approach is faster than the bottom-up approach. But then for the next couple of steps, the bottom-up approach is faster than a top-down approach. And then when we get to the end, the top-down approach becomes faster again.

So the top-down approach, as I said, is more efficient for small frontiers, whereas a bottom-up approach is more efficient for large frontiers. Also, I want to point out that in the top-down approach, when we update the parent array, that actually has to be atomic. Because we can have multiple vertices trying to update the same neighbor.

But in a bottom-up approach, the update to the parent array doesn't have to be atomic. Because we're scanning over the incoming neighbors of any particular vertex v serially. And therefore, there can only be one processor that's writing to parent of v.

So we choose between these two approaches based on the size of the frontier. We found that a threshold of a frontier size of about n/20 works pretty well in practice. So if the frontier has more than n/20 vertices, we used a bottom up approach. And otherwise, we used a top-down approach.

You can also use more sophisticated thresholds, such as also considering the sum of out-degrees, since the actual work is dependent on the sum of out-degrees of the vertices on the frontier. You can also use different thresholds for going from top-down to bottom-up and then another threshold for going from bottom-up back to top-down. And in fact, that's what the original paper did. They had two different thresholds.

We also need to generate the inverse graph or the transposed graph if we're using this

method if the graph is directed. Because if the graph is directed, in the bottom-up approach, we actually need to look at the incoming neighbors, not the outgoing neighbors. So if the graph wasn't already symmetrized, then we have to generate both the incoming neighbors and outgoing neighbors for each vertex. So we can do that as a pre-processing step. Any questions?

OK, so how do we actually represent the frontier? So one way to represent the frontier is just use a sparse integer array, which is what we did before. Another way to do this is to use a dense array.

So, for example, here I have an array of bytes. The array is of size n, where n is the number of vertices. And I have a 1 in position i if vertex i is on the frontier and 0 otherwise. I can also use a bit vector to further compress this and then use additional bit level operations to access it.

So for the top-down approach, a sparse representation is better because the top-down approach usually deals with small frontiers. And if we use a sparse array, we only have to do work proportional to the number of vertices on the frontier. And then in the bottom-up approach, it turns out that dense representation is better because we're looking at most of the vertices anyways. And then we need to switch between these two methods based on the approach that we're using.

So here's some performance numbers comparing the three different modes of traversal. So we have bottom-up, top-down, and then the direction optimizing approach using a threshold of n/20. First of all, we see that the bottom-up approach is the slowest for both of these graphs. And this is because it's doing a lot of wasted work in the early iterations.

We also see that the direction optimizing approach is always faster than both the top-down and the bottom-up approach. This is because if we switch to the bottom-up approach at an appropriate time, then we can save a lot of edge traversals. And, for example, you can see for the power law graph, the direction optimizing approach is almost three times faster than the top-down approach.

The benefits of this approach are highly dependent on the input graph. So it works very well for power law and random graphs. But if you have graphs where the frontier size is always small, such as a grid graph or a road network, then you would never use a bottom-up approach. So this wouldn't actually give you any performance gains. Any questions?

So it turns out that this direction optimizing idea is more general than just breadth-first search. So a couple years ago, I developed this framework called Ligra, where I generalized the direction optimizing idea to other graph algorithms, such as betweenness centrality, connected components, sparse PageRank, shortest paths, and so on. And in the Ligra framework, we have an EDGEMAP operator that chooses between a sparse implementation and a dense implementation based on the size of the frontier.

So the sparse here corresponds to the top-down approach. And dense corresponds to the bottom-up approach. And it turns out that using this direction optimizing idea for these other applications also gives you performance gains in practice.

OK, so let me now talk about another optimization, which is graph compression. And the goal here is to reduce the amount of memory usage in the graph algorithm. So recall, this was our CSR representation.

And in the Edges array, we just stored the values of the target edges. Instead of storing the actual targets, we can actually do better by first sorting the edges so that they appear in non-decreasing order and then just storing the differences between consecutive edges. And then for the first edge for any particular vertex, we'll store the difference between the target and the source of that edge.

So, for example, here, for vertex 0, the first edge is going to have a value of 2 because we're going to take the difference between the target and the source. So 2 minus 0 is 2. Then for the next edge, we're going to take the difference between the second edge and the first edge, so 7 minus 2, which is 5. And then similarly we do that for all of the remaining edges.

Notice that there are some negative values here. And this is because the target is smaller than the source. So in this example, 1 is smaller than 2.

So if you do 1 minus 2, you get a negative-- negative 1. And this can only happen for the first edge for any particular vertex because for all the other edges, we're encoding the difference between that edge and the previous edge. And we already sorted these edges so that they appear in non-decreasing order.

OK, so this compressed edges array will typically contain smaller values than this original edges array. So now we want to be able to use fewer bits to represent these values. We don't want to use 32 or 64 bits like we did before. Otherwise, we wouldn't be saving any space.

So one way to reduce the space usage is to store these values using what's called a variable length code or a k-bit code. And the idea is to encode each value in chunks of k bits, where for each chunk, we use k minus 1 bits for the data and 1 bit as the continue bit. So for example, let's encode the integer 401 using 8-bit or byte codes.

So first, we're going to write this value out in binary. And then we're going to take the bottom 7 bits, and we're going to place that into the data field of the first chunk. And then in the last bit of this chunk, we're going to check if we still have any more bits that we need to encode. And if we do, then we're going to set a 1 in the continue bit position.

And then we create another chunk. We'll replace the next 7 bits into the data field of that chunk. And then now we're actually done encoding this integer value. So we can place a 0 in the continue bit.

So that's how the encoding works. And decoding is just doing this process backwards. So you read chunks until you find a chunk with a 0 continue bit. And then you shift all of the data values left accordingly and sum them together to reconstruct the integer value that you encoded.

One performance issue that might occur here is that when you're decoding, you have to check this continue bit for every chunk and decide what to do based on that continue bit. And this is actually unpredictable branch. So you can suffer from branch mispredictions from checking this continue bit.

So one way you can optimize this is to get rid of these continue bits. And the idea here is to first figure out how many bytes you need to encode each integer in the sequence. And then you group together integers that require the same number of bytes to encode.

Use a run-length encoding idea to encode all of these integers together by using a header byte, where in the header byte, you use the lower 6 bits to store the size of the group and the highest 2 bits to store the number of bytes each of these integers needs to decode. And now all of the integers in this group will just be stored after this header byte. And we'd know exactly how many bytes they need to decode.

So we don't need to store a continue bit in these chunks. This does slightly increase the space usage. But it makes decoding cheaper because we no longer have to suffer from branch mispredictions from checking this continue bit.

OK, so now we have to decode these edge lists on the fly as we're running our algorithm. If we decoded everything at the beginning, we wouldn't actually be saving any space. We need to decode these edges as we access them in our algorithm.

Since we encoded all of these edge lists separately for each vertex, we can decode all of them in parallel. And each vertex just decodes its edge list sequentially. But what about high-degree vertices? If you have a high-degree vertex, you stop to decode its edge list sequentially. And if you're running this in parallel, this could lead to load imbalance.

So one way to fix this is, instead of just encoding the whole thing sequentially, you can chunk it up into chunks of size T. And then for each chunk, you encode it like you did before, where you store the first value relative to the source vertex and then all of the other values relative to the previous edge. And now you can actually decode the first value here for each of these chunks all in parallel without having to wait for the previous edge to be decoded. And then this gives us much more parallelism because all of these chunks can be decoded in parallel. And we found that a value of T-- where T is the chunk size-- between 100 and 10,000 works pretty well in practice.

OK, so I'm not going to have time to go over the experiments. But at a high level, the experiments show that compression schemes do save space. And serially, it's only slightly slower than the uncompressed version. But surprisingly, when you run it in parallel, it actually becomes faster than the uncompressed version.

And this is because these graph algorithms are memory bound. And we're using less memory. You can alleviate this memory subsystem bottleneck and get better scalability. And the decoding part of these compressed algorithms actually gets very good parallel speedup because they're just doing local operations.

OK, so let me summarize now. So we saw some properties of real-world graphs. We saw that they're quite large, but they can still fit on a multi-core server. And they're relatively sparse.

They also have a power law degree distribution. Many graph algorithms are irregular in that they involve many random memory accesses. So that becomes a bottleneck of the performance of these algorithms. And you can improve performance with algorithmic optimization, such as using this direction optimization and also by creating and exploiting locality, for example, by using this bit vector optimization.

And finally, optimizations for graphs might work well for certain graphs, but they might not work well for other graphs. For example, the direction optimization idea works well for power law graphs but not for road graphs. So when you're trying to optimize your graph algorithm, we should definitely test it on different types of graphs and see where it works well and where it doesn't work.

So that's all I have. If you have any additional questions, please feel free to ask me after class. And as a reminder, we have a guest lecture on Thursday by Professor Johnson of the MIT Math Department. And he'll be talking about high-level languages, so please be sure to attend.