PROFESSOR: Thanks for coming. I know there's a problem set due. There's a quiz coming up on Thursday. We won't have lecture on Thursday. But we will have a quiz in the evening. And there will be a recitation section tomorrow, which will be a quiz review.

So, today's lecture is the last of the lectures in the shortest path module. And, unlike previous lectures, we're going to be talking about optimizations that don't change the worst case, or asymptotic, complexity. But improve empirical, real life performance. Or potentially, and we can't prove this, but performance in the average case.

And so we look at a couple of examples. The first one you've already done. You can optimize Dijkstra when you're looking for a single target.

So, implicitly, we've assumed that we're solving the single source, any or all destination problem, when we've looked at the original Dijkstra algorithm, and the Bellman-Ford algorithm. Many a time, you're going to have a source, s. And you want to find the shortest path to a specific destination, t.

And you're doing this in your problem set. And you can do some optimization. It doesn't change the worst case complexity. But it reduces runtime. And then you have a specific target. Especially if the target is close to you. And you don't have to traverse the entire graph.

Obviously, the algorithm has to prove that the particular path that was chosen is, in fact, the shortest path from s to t. But it's a fairly straightforward modification. And I will go over it, but you're actually implementing it in PS 6.

We talked about bi-directional search. Again, something you're doing from a

standpoint of breadth-first search. How can you get from one source to a destination, by doing bi-directional Dijkstra?

And you can think of this as a frontier off shortest paths is being constructed from the source, s, forward. And this backward frontier, you're falling edges backward, is being constructed from the destination. And, effectively, when these two frontiers meet, you're going to be able to discover shortest paths.

And, it turns out, it's not as simple as what I just described. And so we'll have to look at that a little more carefully. So that is our single source, single target problem. We won't cover this in 006.

But there's also this notion of all pairs shortest paths, which is something that 6046 covers. And that is something that says, well, I don't know what the source is. I don't know what the destination is. For any pair of vertices, find me the shortest path from s to t, given that pair.

And so that, of course, is more work than even the single source, all destination problem because you're varying the source. All right? So those are the three different shortest path problems.

And we've looked at that. And we're going to look at this today. And we looked at it in the problem set. But we'll spend time, specifically on this one. And try and see if we can do some coding optimizations, if you will, to improve run time.

All right? So, I emphasize that worst case complexity is unchanged for all of the Dijkstra versions that we'll be looking at today. So I want to put up a pseudocode that you've written code for at this point, which is the Dijkstra pseudocode. Because we'll take a look at it, and modify it, and execute it.

So you have your set of vertices that you don't know the shortest paths to. So we do have initialize is going to set d of s to be 0. And b of u, not equal to s, to be infinity.

And we have this set, Q, that we're going to process and continually extract the minimum priority from Q. And, once we do that, we actually know the shortest path

to you already. That's what the Dijkstra invariant is.

And the process of extracting u implies that we have to do a relaxation step that updates the priorities. And also modifies the parent pointers. So there's also the pi of v that is set to u. As well as d of v getting decremented. OK? So that's the Dijkstra algorithm.

And one of the things that we can do, the straightforward thing, which is one line of code literally, is to say that, if you know what the single target is, then you simply stop if u equals t. So no need to stop when Q becomes null. Or, you don't get to the point where Q is null.

You stop when you've lifted off the y vertex from Q. All right? And so, obviously, this will run faster, assuming this check is a 0 time check. And that's really one instruction, so you can think of it that way. And you will, basically, run faster for sure, when you have a specific target.

It may be the case that your target is the last vertex that you find. And in that case, you run no slower. All right? So that's something that you looked at. And that takes care of the first optimization corresponding to single source, single target.

Let's talk about something that's a little more interesting, and non-obvious, which is the notion of bi-directional search. And, in bi-directional search, we have s. And we have t. And we have a bunch of nodes in between, corresponding to this graph here.

And what you do is, you alternate search in the forward direction and backward direction. So, you're going to do one step of Dijkstra, standard Dijkstra that starts with s. And goes forward.

And so, you could imagine that you're forward search in the first step, you're going to pick the vertex, s, off of Q. And you're going to process the edges that come out of s. And that would correspond with these two edges. And so that's one step of forward search, going forward.

And then you stop with the forward search, and you do a step of backward search. And so, you go backward search, and I'll explain exactly what this means, backward search from t. And the important thing is that you're following edges backward.

So that means your data structure has to, essentially, have these edges that can be traversed in either the forward direction, or the reverse direction. So that's something to keep in mind. But what will happen here is simply that your first frontier of backward search, t now, in the backward search, is the minimum priority.

So , we're going to have to have two priorities corresponding to s in the fourth search, where that's a straightforward one. v of s equals 0. And we should think of it as d of f s equals 0. And we have d of b t equals 0. And these subscripts correspond to these two different priorities. And it's exactly complimentary.

Only the source, s, in the forward search, has 0 priority in the beginning. Everything else has infinite priority. Only the target, or destination, has 0 priority in the backward search. Everything else is infinity.

And you go forward, backward, forward, backward. And so on. And the question is, when do you stop? And we have to talk about that. And, it turns out, it's not a perfectly straightforward stopping condition. But it's something that will make sense, hopefully, when I get around to describing it.

But, having said all that, let's just put down all of the different data structures that we have to have. And it's kind of a doubling of the data structure, right, because just like I double the priorities, I also need two different min priority queues, corresponding to Q f and Q b. And, as I said before, these edges have to be traversable in the backward direction.

So this edge, in the graph, goes this way. But you're going in the backward direction, as you are growing your backward frontier. OK? That's important to understand. So let me put down all of the specifics associated with these data structures.

df u correspond to the distances for the forward search. And db u correspond to the

distances for the backward search. OK? And, of course, we're going to have to have priority queues, plural.

Qf corresponding to the forward search. And Qb corresponding to the backward search. And in initialize, as I said before, we're going to initialize df s to be 0 and db t equals 0. And then everything, the df's and the db's, are going to be infinity.

All right? OK. Great. So that's what we have here. Now, first question. Roughly speaking, as I said, you can imagine intuitively that you're going to terminate the search when these frontiers meet, OK? Clearly, you can't terminate it when these frontiers look like this, OK. So that's the intuition.

And you see that for the [INAUDIBLE] first, as well, in your problem set. But in the context of Dijkstra, single source, single target, can someone tell me what the termination condition should be by looking at the code? I want a more specific, or a more concrete, termination condition that I can actually code up, as opposed to saying, the frontiers meet. Which, you know, I don't know how to code. OK? Someone else? All right, go for it.

AUDIENCE: When there's some node that can keep track of two different cost values from--

PROFESSOR: The Qf and the Qb, that's correct. And, somehow--

AUDIENCE: Somehow they're cost runs to get there from the start and from the [INAUDIBLE].

PROFESSOR: OK. It's close. It's not quite something I can code up. Someone want to improve that? Someone want to improve that? I want something very, very specific. Someone? Go for it.

AUDIENCE: The node has been extracted from both Qf and Qb.

PROFESSOR: The node which has been extracted from both Qf and Qb. So, the reason I didn't quite buy your answer was, finite part is obviously correct. But I wanted a specific condition that says, I'm going to do extract-min, just like I said when I extract-min and u equals t, I stop with the single source, single target.

In the bi-directional case, I need to pull out a node from Qf. And pull out a node from Qb. And then I get to stop, all right? So, you get a cushion.

You don't need to feel too bad because I think you already have a cushion. Yeah. I know that. I know everyone who has cushions. Right. Actually, I don't. But I'm going to pretend I do.

So, the termination condition is that some vertex, and this is correct, some vertex, u, has been processed, both in the forward search and the backward search. OK? That corresponds to the frontiers meeting. But, specifically, it's been deleted, or extracted from both Qf and Qb.

So that's actually the easier question. Visit a harder question, which is how do we find the shortest path after termination from s to t? OK? And I should say specifically, that-- and I forgot to put this up, which I should-- that we're going to have to have pi f and pi b, which this is the normal data structure.

And the pi b is following the edges backward. So, in some sense, the predecessor, in the case of pi b, what you're saying is, on this node here-- which I'll call v2 for example-- is, if I'm going to choose this path here-- and I need to obviously choose this edge here in any shortest path that gets to t, right?

Because that's the only edge that goes to t. And so, what I'm saying here is that a predecessor of pi b t equals v2. OK? That make sense?

And then over here, if this was v1, then I would have pi f v1 equals s. Right? That make sense? Everybody buy that? All right, so how do I find the shortest path from s to t, after these frontiers have met, and I've terminated the search? How do I do that? Someone? Go ahead.

AUDIENCE: Well, given that data structure, you start at your t. And you keep on going back to the [INAUDIBLE] until you get to the point where they've peaked. Then you use the pi f to go from that node, all the way back to s.

PROFESSOR: So, what I'm going to have to do is-- but where do I switch, is the question. Where do I switch from pi f to pi b? There has to be some point where I switch from-- Yeah, go ahead.

AUDIENCE: At the meeting point of the node that was [INAUDIBLE] Q [INAUDIBLE].

PROFESSOR: All right. Is that what you were saying, too? OK. So the claim is, if w was processed first, extracted from both Qf and Qb, then find the shortest path using pi f from s to w. Right? So, you can use pi f to get from s to w.

And the way you do that is by applying pi f to w. And then keep applying it until you get to s, OK? This is normal search, right? Everybody knows this. You coded it. So I hope you know it.

And then, we go find shortest path using pi b, right? And you're going to constantly apply pi b-- and this is the backward path-- from t to w. And this follows the edges backward, all right? This sounds pretty good? Everybody agree with this? Anybody disagree? Yeah.

AUDIENCE: Pi b. If [INAUDIBLE], pi b would be like pi b b2 equals t, because b2 [INAUDIBLE]--

PROFESSOR: That's a good question. I might have done this wrong. So, in the backward search, this can get pretty confusing. So what do I have here? I want to follow the predecessor. You're exactly right. You're exactly right. Thank you. Thank you for pointing that out.

All right, so what I have here is when I look at this path that goes this way, all right? I'm going to look at the path that goes this way. s is the predecessor of v1. v1 is a predecessor of, let's call this v3. v3 is a predecessor of v4. I'm just talking about the regular forward path.

We have s to v1. v1 to v3. v3 to v4. All the way to t, right? So what I have here is correct. The predecessor of v1 is s. The predecessor of v3 would be v1. So I could write, pi f v3 equals v1. Et cetera. Now, let's just forget about the forward path, and let's just talk about the backward path.

In the backwoods path, I want to be able to construct this backward path. It's got to be the reverse of what I have, OK? And, in that case, what I'm saying is that I want to move in this direction. So pretend that I've made the edges flip, OK?

So, in that case, if I pretend that this edge is like that, and then I just apply the regular predecessor relationship, then t is the predecessor of v2. And that's the point you're making. OK? t is the predecessor of v2. And so, my apologies.

I have pi b v2 equals t. And, if I follow this edge here, then I'd have the appropriate relationship. But let's just stick to this one because that's the simple example.

I don't quite know whether this edge is going to be part of my shortest path or not. It might be. And that's something that we'll compute. But what I have here is the predecessor relationship corresponding to the backward edge.

And so, that's like flipping this edge. And, hopefully, that makes sense now. Thanks for pointing that out.

And so, let's talk about what happens here. We know what happens in the forward search. You've done that before. In the backward search, what happens is that I need to start-- according to this condition-- just like in the forward search, I found a w. And I continually applied pi of f to w.

So, this is apply pi of f to w. And then do pi of f pi of f w. And so on and so forth. And that's what you do in order to construct the shortest path. People buy that? Right?

And what I want to do here is apply pi b to w. And then, pi b of pi b to w. And so on and so forth, till I get to t. Right? And this one, till I get to s. Right?

So, what I wrote here, s to w, t to w. There's nothing incorrect about that. What's important to understand is the application of the pi f and the pi b. Both, according to this, start with w, which is this vertex that caused the termination to happen. All right?

So, people buy this? Any other questions? All right. Turns out this is not quite correct. OK? This is not quite correct, right? And not because of the pi b inversion that I had before, right?

So what have I said, so far? It makes perfect sense. It says, I have a vertex that caused the termination. I'm going to call it w. OK? And that vertex is on the intersection of these two frontiers, OK?

And I'm going to use that to construct the shortest path by constructing two sub paths, using the forward pointers and the backward pointers. All right? So all of that makes sense except, it turns out, that w may not be on the shortest path. OK?

And I'll show you an example where w is not on the shortest path. All right? So that's at a real subtle condition. So we have to actually augment the termination condition. Or, we have to do something more than the termination condition.

So, I will tell you right away, the termination condition is correct. OK? And so, the guy who got the cushion, deserved the cushion. OK? So the termination condition as correct.

You are going to run Dijkstra 's ultimate forward search and backward search. And you're going to terminate when a particular vertex, call it w, is going to get pulled out from both Qf and Qb. All right? What is incorrect here is the use of w to construct the shortest path. All right?

It turns out, we have to do a little more work to go find the shortest path, after we've terminated. And w may not be on the shortest path. All right? Any ideas as to what we might do? This is a bit of an unfair question, but certainly worth a cushion.

How do you think we can fix this? If w is not on the shortest path, what do you think would be on the shortest path? Is there a way of finding this vertex, so we can break this up?

We absolutely have to use both pi f and pi b. There's no getting away from that because these two frontiers have just barely collided. The instant they barely

collided, we've stopped. OK?

So we can't use pi f all the way from s to t. We can't use pi b all the way from t to s. These frontiers have just barely collided. OK? So what happens if w is not on the shortest path? And why is that the case? Yeah.

AUDIENCE: I just had a question. Are all the edge weights identical? Or are they--

PROFESSOR: So, the edge weights don't change. There are no new edges. The way you want to think about is that, you can traverse the edges backward. And so, it's not like there are two edges here.

Now, you could fake it, and have two edges with exactly the same weights over here. But are you saying there are edge weights in the graph, all identical across the edges? Or are you asking about the forward search versus the backward search?

AUDIENCE: That's what I was asking. Is each edge weight the same in the graph.

PROFESSOR: No. They're using Dijkstra. The edge ways can be arbitrary. But they're non-negative. OK? So that's the usual Dijkstra requirement. They could be real numbers. They could be irrational numbers. They could be whatever. But they're all non-negative. All right?

Now, in the backward search and the forward search-- just to make that clear-- while I've drawn this particular edge, that weight hasn't changed, OK? That weight had better be the same. OK? All right, so I'll show you an example.

And we'll take an example, a fairly straightforward example, it turns out. It took a while to concoct this five node example that shows the idea here. But what we're going to do is, take a look at the termination condition in a specific case, where we're going to do this alternation of forward and backward search. And we'll see, when it terminates, as to what the correct way is to construct the shortest path. All right?

And, as I said before, the termination condition is correct. It's not like we stop too early. When one of those nodes gets off from Qf and Qb, you get to stop.

So that's my s, over here. And I have a fairly straightforward graph. I have 5, 5, 3, 3, 3. So we don't need a computer program to tell us that the shortest path from s to t is the path with three edges that goes on top, OK, which has a weight of 9.

All right. So this is a forward search. And I'm going to call all of these vertices names. So I have u prime. t. et cetera. OK?

So, in the first step of the forward search, I'm going to be able to set-- oh, I'm sorry. This one is a w. I'm going to be able to set df w equals 5. And df u equals 3. And, obviously, df of s equals 0. And I'm not going to bother writing the infinities. It's just going to clutter up the board, all right?

So, stop me if you have questions on anything I'm writing here. So that's a forward search. Now, let's do the first step of backward search, right? Alternate, remember?

Alternate forward search, backward search, forward search, backward search. And I'm just going to write this out again, so bear with me, because I think it'd be clearer if you see this graph many times. As opposed to my erasing what I've written.

So I've got an s, here. t there. u. u prime. w. And I'm going to hash this vertex vertically because that's my backward search. And db of t equals 0. OK? And I'm going to follow this backward, and this backward.

And my weights are the same. It's the same graph. So I'm going to have d of b u prime equals 3. And db of w equals 5. And I haven't seen u yet. I haven't seen s yet. And so, all I've done is mark these two. All right?

So far, so good? Again, stop me if you have questions. We've got, obviously, a couple more steps to go here. And let's keep going. So now we do a forward search again.

Yeah, that's fine. That's this hash that way, just to make sure. This is s, u, u prime, w, t. And what I have now is, I'm going process this vertex in the forward search

because I have a choice in the forward search to either pick w or w.

This clearly has lower priority because df w equals 5. And df of u equals 3. So extract-min is obviously going to pick u. And it's going to process this edge now, after extract-min. And I'm going have df of u prime equals 6. OK?

AUDIENCE: It's 3 over there.

PROFESSOR: Oh, it's 3 over there. Thanks. Good. So, so far, so good? Yeah? All right. So now, I go to the backward search. And again, I have s, t, u, u prime, w.

I'm going to go ahead and hash this. This has been hashed horizontally. The hash horizontally means that it's been removed from Qf. The hash vertically means that it's been removed from Qb.

And so, when I look at this, and I do a backward search, I'm going to hash this. And I'm going to set db of u prime equals 3. And I have db of w equals 5. So, that I already had.

And so, when I have db of w equals 5 and db of u prime equals 3, then, obviously, I'm going to pick the one with the min priority. This corresponds to this one. And what it's going to do is, it's going to go process that and set db of u equals 6. All right?

So what happened here simply was that I picked this vertex off of Qb because that was the min priority. And all I did was relax this particular edge in the backward direction. And said db u equals 6.

All right? Almost there. Any questions so far? Any bugs you noticed so far, in what I've written? Yeah, back there, question.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. Good. So people agree with this. All right, one more. One more board to draw. And we'll be done. We will have terminated. All right, so getting pretty close.

So I'm set up this way. Oh, shoot. Sorry. This is horizontal. So, now I'm looking at it, and I've taken care of these two. And I'm talking about the forward search here. So this is, again, the forward. And that's the backward.

And now I'm doing a forward again. And my only choice, now, is to pick the w vertex off of Qf. Right? Because df w equals 5. And this one has already been processed. df of u equals 3. But I've hashed that. And so I've gone ahead and processed that vertex.

And this one, df of u prime equals 6. So, in Qf, I would be comparing u prime and w. And I would take w, OK? People buy that? That's because, I guess you by the fact that 5 is less than 6. I hope.

So that's what happens in the step of forward search. And then you go ahead and process this. You're going to set df of t to be 10. OK?

And now you're starting to see why there may be a bit of a problem with our shortest path computation, right? Maybe. All right? Everything good? All right.

So what have I done here? I've removed w from Qf. OK. I've removed w from Qf. All right. Now, let's look at the last step here, of the backward search.

s, t. And so, this was hashed. That was hashed. And, if I look at what I have here, I have db u prime equals 3. df u prime equals 6. This was df w equals 5. db w equals 5. And so on and so forth.

Again, you compare w. And you see that db w equals 5. And df u prime equals 6. So therefore, you will pick w. OK?

You will pick w, and remove it from Qf. So remove w from Qf. All right? And process it. And what you end up with is df of s equals 10. OK?

That's what you get because this is a 5. And that's a 5. OK? People see the problem here? What's the problem? Someone articulate the problem.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Yeah. I mean, so, what I have so far is, I've terminated. But it looks like I end up with d of s. If I look at it from a standpoint of the forward weight, I got a 10 for t.

I get db t equals 10. If I look from a standpoint of the backward weight, I get db s equals 10. OK? And we all know that the shortest path should be 9. OK.

So what happened here? Well, we terminated according to this condition. We terminated when w was pulled off from Qf and Qb because that was the short path, in some sense, in terms of number of edges, right?

It was only of length 2. And so, then the frontiers collide. This is a subtlety in the algorithm. The frontiers collide at some vertex, regardless of the weights of the edges. Because we are alternating the forward search and the backward search, in effect, the frontiers are going to collide on the shortest length path. Right? That makes sense, right?

So this example is set up so the shortest length path is not the shortest weight path. OK? So we can't take the w and use pi f to construct part of the path, and use pi b to construct the other part of the path. And find the shortest weight path. We would get something incorrect. Right? We would get a path of length 10, in this case, if we did that.

So how do we fix it? How do we fix it? One little termination condition doesn't change. How do we fix it? Someone? Back there. Or, actually, you.

AUDIENCE: Ordinate based on the weight.

PROFESSOR: Ordinate based on the weight. So exactly how would we do that?

AUDIENCE: At each point you take the one that has the [INAUDIBLE].

PROFESSOR: But that's what we're doing now, right? So, are you changing the alternation between the forward search and the backward search?

AUDIENCE: Yeah.

PROFESSOR: You're saying that you're going to do more forward searches, as opposed to backward searches?

AUDIENCE: If the overall cost is lower.

PROFESSOR: Overall cost is lower. You know, if you code it up, you get to keep this. If I want a slightly different, simpler fix because I think what you're saying here-- and I like the idea. I actually do like the idea.

You're saying you're going to not do strict alternation. But you're going to do some sort of weighted alternation, from what I can tell, based on the weights. OK? And I think there's an algorithm there that's correct. OK?

I probably won't be able to prove that it's correct to myself in five minutes. OK. Or 10 minutes. But let's talk about that offline. And see if there's a way.

I'm a little worried that, if you have positive rational numbers, and you've got root 2, square root of 2, and pi. And the transcendental number corresponding to these weights, that this weighted alternation is going to be a bit hard to implement correctly. But it's a neat idea. I would actually like a follow up on that. Something that's a little-- yeah, back there.

AUDIENCE: Can you look at the neighbors of all the nodes that are in your forward [INAUDIBLE], and see if any of their neighbors are in backwards [INAUDIBLE]. And see if that's going to give you a shorter path than the one that you pulled out previously.

PROFESSOR: That's correct. That's almost exactly correct. I won't bother throwing it over, but this is yours. You want to catch it?

AUDIENCE: I already have one.

PROFESSOR: You already have a cushion. All right. So how about I just throw it. And anybody who wants to catch it gets it. How's that? Whoa.

All right. I didn't hit anybody. Anybody who wants it, who doesn't have a cushion. Right. We've got all these cushions in my office, and Eric's office. We've got to do something about them.

They're actually not that great. You know, it's bit of an issue, I understand. You know, we've got to do Frisbees next time, or something. I can see why you're not enthusiastic about this. Right? OK.

All right, so the answer was, in fact, correct. And let me write it up over here. So we had to do a little more work. We terminate properly. We do the strict alternation. But we have to do a little bit more work. OK.

And the work we have to do is summarized very neatly by saying, we want to find an x, which is maybe different from w, possibly, that has minimum value of d of x plus db x. All right? So we have to actually look. And this x is going to be neighboring.

But we don't really need to specify that. What we say is, you're going to have to look at Qf and Qb. And it's possible that w is the one that has minimum df w plus db w. But, clearly, that wasn't the case in this example.

So, in this example, w caused the termination. But now we need to scan. And there's only two other vertices that are interesting here, which are u and u prime.

And either of those will fit the bill because we see that df of u plus db of u equals 3 plus 6 equals 9. And df of u prime plus db of u prime equals-- I'm sorry, I should have out db here. db of u prime would be 6 plus 3 equals 9. Right? So both of these are less than 10.

And so we had to pick one of these as our x. OK. And if you pick one of these as our x, then, at that point, we don't do w here. We do x. OK? Let me write it as x. It looks like a cross here.

So that's what happens. So this small little tweak. Do the alternation. Do the termination. Once you do the termination, go look and see if you can minimize the shortest path length by finding an appropriate x.

Which has the minimum sum of the forward and the backward priorities. And then, you're in business. And then, everything works. OK?

All right. Great. So, so much for bi-directional search. Let me talk a little bit about heuristics that people use to modify the graph so things run faster in practice. So, in particular, you can think about the goal directed search, or a star, if you're taking 6034. And there's some commonality between what I'm going to talk about here, and that material.

The basic idea is that we're going to modify the edge weights in such a way that you go downhill toward the shortest path. And so, the priorities are modified heuristically. So things run a little bit quicker. You're trying to prune the search here.

So we're going to modify edge weights. And we have to be careful when we do this, obviously. We don't want to do things that are incorrect.

But the way we want to modify the edge weights is by having some sort of potential function that corresponds to lambda. And, if the edge is between u and v, then we have an equation given a lambda that says, the new w, w bar, is w u, v minus lambda u plus lambda v. OK?

Now, we have to be a little careful here, with respect to the choice of lambda. But the basic idea, in terms of lambda, is that, suppose you have something like this, where you have a source vertex, s. And you're again trying to do a single source, single target going to t.

And let's say I have an edge of weight 5 going out and an edge of a 5 going out this way. Is there a way that you can guess that this is the edge that is more likely to be on your shortest path? As opposed to this other edge?

Essentially increase the potential of this node, all right? So this node here may be the node t2. And this node may be the node t1.

You want to increase the potential of node t2, such that, you're actually trying to go uphill when you go this way. And this goes downhill. And that has the appropriate

modifications on the edge weights, such that the Dijkstra algorithm is steered towards going downhill. And going down this path. And it terminates a little bit quicker. Right?

It doesn't change as asymptotic complexity. It just makes things run, in practice, a little bit faster. If you choose the right potentials. Right? Feels like magic.

How do you know how to increase the potential? What would you increase the potential for? What nodes do you want to make uphill? What nodes do you want to make downhill?

So there's a bunch of questions. I'm not going to get into a lot of details. But I will tell you a couple of things.

I'm going to give you, really quickly, a simple example that is both correct, in terms of the actual shortest path you will get is the correct one. And a particular mechanism off modifying the potentials that uses landmarks. Right. So the way we are going to do this is by saying that any path wp is going to get modified based on its destination and source.

So the only way that we can use the potential method is by ensuring that all of the shortest paths between any pair of vertices, we're only concerned about single source, single target here. But, in general, it's a good thing to not change any of the shortest paths. So what used to be a shortest path should stay the shortest path.

And the way you do that is by having a potential function that, if you have an arbitrary path, essentially-- and this is a path from s to t. That you subtract out something based on a function of the vertex. In this case, you have s. And, in this case, you have t.

So the nice thing is that, any path from s to t is going to get shifted by the same amount, corresponding to this additional term here. So what that means is that the final shortest path that you discover will be the correct shortest path. You just may, hopefully, discover it faster. All right?

So that's the correctness check. And I'll put this in the notes. And maybe the TAs can cover it in the section. But one way of getting this potential function is to use what's called a landmark.

And so the basic idea is that you have a landmark, l, which is a vertex belonging to v. And we're going to pre-compute delta of u comma l. So, for any input vertex, you want to find the shortest path to this landmark.

So it's like, change the source, but the destination stays the same. And the potential lambda t u is defined as delta of u, l minus delta of t, l. OK? So you have the source, s. You have a destination, t. And now you have a landmark, l.

I'm going to pre-compute delta u,l for all u belonging to v. And I'm also going to pre-compute, for a given t, delta t, l. So that's just a single t. So that's just one computation. This one is much more computation.

And, if I use this potential, you can show that it's correct, using the triangle inequality. And this is not a heuristic. With the correct choice of landmark, Dijkstra, empirically, will run faster.

So, if you know for sure that you need to go through middle America to get from Cal Tech to Boston-- and there's one particular landmark you want to go through-- Texas or something. And you pick Austin, Texas, then you can do this computation.

And maybe Dijkstra runs 2x faster, 20% faster. All right? I'll put the argument about correctness, and the specifics of these things, in the notes. And you can take a look at it offline.