

wk6

2-wk2



wk6

# 0wk6-Overview

SUMMARY: This week is the climax of all our work on graph search --- Dijkstra's shortest-path algorithm, surely one of the greatest hits of algorithms (Part XI). It works in any directed graph with non-negative edge lengths, and it computes the shortest paths from a source vertex to all other vertices. Particularly nice is the blazingly fast implementation that uses a heap data structure (more on heaps next week).

THE HOMEWORK: Problem Set #2 should solidify your understanding of Dijkstra's shortest-path algorithm. In the programming assignment you'll implement Dijkstra's algorithm. You can just implement the basic version, but those of you who want a bigger challenge are encouraged to devise a heap-based implementation.

SUGGESTED READINGS FOR WEEK 2:

CLRS Chapter 24 (Sections 3,4)

DPV Sections 4.4

KT Section 4.4

SW Section 4.4

# 11-1-Dijkstra's Shortest-Path Algorithm

We've arrived at another one of computer science's greatest hits, namely Djikstra's shortest path algorithm. So let me tell you about the problem, it's a problem called single source shortest paths. Basically what we want to do is compute something like driving directions. So we're given as input a graph, in this lecture I'm going to work with directed graphs, although the same algorithm would work for undirected graphs with cosmetic changes. As usual, we'll use m to denote the number of edges and n to denote the number of vertices. The input also includes two extra ingredients. First of all, for each edge e we're given as input a non-negative link which I'll denote as l sub e. In the context of a driving directions application l sub e could denote the mileage how long this particular road is, or it could also denote the expected travel time along the edge. The second ingredient is a vertex from which we are looking for paths. This is exactly the same as we had in breadth first search. In that first search we have an originating vertex which we'll call here the source. Our responsibility then is to, given this input, to compute for every other vertex, v, in this network the length of the shortest path from the source vertex, s, to that destination vertex, v. And so just to be clear, what is the length of the path that has say three edges in it? Well it's just the sum of the link of the first edge in the path plus the length of the second edge in the path plus the length of the third edge in the path. So if you had a path like this with three edges and length one, two and three, then the length of the path would just be six. And then we define the shortest SV path in the natural way, so amongst all of the paths directed from S to V, each one has its own respective path length and then the minimum overall SV paths is the shortest path distance in the graph G. So I'm going to make two assumptions for these lectures. One is just really for convenience, the other is really important. The other assumption, without which, Dijkstra's algorithm is not correct, as we'll see. So for convenience we'll assume that there is a directed path from S to every other vertex V in the graph, otherwise the shortest path distance is something we define to be plus infinity. And the reason this is not a big assumption is, if you think about it, you could detect which vertices are not reachable from S just in a preprocessing step using, say, breadth-first or depth-first search. And then you could delete the irrelevant part of the graph, and run Dijkstra's algorithm as we'll describe it on what remains. Alternatively, Dijkstra's algorithm will quite naturally figure out what vertices there are paths to from S and which ones there are not, so this won't really come up. So to keep it simple, just think about we have an input graph where you can get from S to V, for every different vertex V. And the challenge then is amongst all the ways to get from S to V, what is the shortest way to do it? So the second assumption already appears in the problem statement, but I want to reiterate it just so it's really clear. When we analyze Jackson's algorithm, we always focus on graphs where every length is non-negative, no negative edge lengths are allowed. And we'll see why a little bit later in the video. Now in the context of a driving directions application it's natural to ask the question why would you ever care about negative edge lengths. Until we invent a time machine that doesn't seem like negative edge lengths are going to be relevant when you are computing literal paths through literal networks. But again remember that paths can be thought of as more abstractly as a just sequence of decisions. And some of the most powerful applications of shortest paths are coming up with optimal weight such sequences. So, for example, maybe you're engaging in financial transactions and you have the option of both buying and selling assets at different times. If you sell then you get some kind of profit and that would correspond to a negative edge length. So there are quite interesting applications in which negative edge lengths are relevant. If you are dealing with such an application, Dijkstra's algorithm is not the algorithm to use. There's a different shortest path algorithm, a couple other ones. But the most well-known one is called Bellman-Ford. That's something based on dynamic programming, which we may well cover in a SQL course. Okay, so for Dijkstra's algorithm, we always focus on graphs. That'll have only non-negative edge lengths. So, with the next quiz, I just want to make sure that you understand the single source shortest path problem. Let me draw for you here a simple four node network, and ask you for, what are the four shortest path lengths. So from the source vertex s, to each of the four vertices in the network. All right, so the answer to this quiz is the final option, 0,1,3,6. To see why that's true, well, all of the options had 0 as the shortest-path distance from s to itself. So that just seemed kind of obvious. So the empty path will get you from s to itself and have 0 length. No suppose you wanted to get from S to V, well there's actually only one way to do that, you have to go along this one hop path. So the only path has length of one, so the shortest path distance from S to V is one. Now W's more interesting, there's a direct one hop path, SW, that has a length of four, but that is not the shortest path from S to W Inf act to two-hop path that goes through v as an intermediary has total path length three which is less than the length of the direct arc from s to w. So therefore the shortest distance from s to w is going to be 3. And finally for the vertex t there's three different paths going from s to t. There's the two-hop path that goes through v. There's the two hop path which goes through w, both of those have path length 7, and then there's the three hop path which goes through both v and w. And that actually has a path length of one plus two plus three equals six. So despite having a largest number of edges, the zigzag path is, in fact, the shortest path from s to t and it has length 6. All right, so before I tell you how Dijstrka's algorthin works, I feel like I should justify the existence of this video a little bit. All right? Because this is not the first time we've seen shortest paths. You might be thinking rightfully so. We already know how to compute shortest paths. That was one of the applications of breadth first search. So the answer to this question is both yes and no. Breadth first search does indeed compute shortest paths. We had an entire video about that. But it works only in the special case where the length of every edge of the graph is one. At the moment we're trying to solve a more general problem. We're trying to solve shortest paths, when edges can have arbitrary non-negative edge lengths. So for example, in the graph that we've explored in the previous quiz, if we ran breadth first search, starting from the vertex s, it would say that the shortest path distance from s to t is 2 and that's because there's a path with two hops going from s to t. Put differently, t is in the second layer emanating from s. But as we saw in the quiz, there's not in fact a shortest two hop path from s to t if you care about the edge lengths. Rather the minimum length path, the shortest path, with respect to the edge weights, is this three hop path which gives us a total length of 6. So breadth first search is not going to give us what we want when the edge lengths are not all the same. And if you think about an application like driving directions, then needless to say, it's not the case that every edge in the network is the same. Some roads are much longer than others, some roads will have much larger travel times than others, so we really do need to solve this more general shortest path problem. Similarly, if you're thinking more abstractly about a sequence of decisions like financial transactions, in general different transactions will have different values. So you really want to solve general shortest paths, you're not in the special case that breadth-first search solves. Now, if you're feeling particularly sharp today, you might have the following objection to what I've just said. You might say, eh, big deal. General edge weights, unit edge weights, it's basically the same. Say you have an edge that has length three. How is that fundamentally different than having a path with three edges, each of which has length one? So why not just replace all the edges with a path of edges of the appropriate length? Now we have a network in which every edge has unit length and now we can just run breadth-first search. So put succinctly, isn't it the case that computing shortest paths with general edge weights reduces to computing shortest paths with unit edge weights? Well, the first comment I want to make is I think this would be an excellent objection to raise. And indeed, as programmers, as computer scientists, this is the way you should be thinking. If you see a problem that seems superficially harder than another one, you always want to ask, well, maybe just with a clever trick I can reduce it to a problem I already know how to solve. That's a great attitude in general for problem solving. And indeed, if all of the edge lengths were just small numbers, like 1, 2, and 3 and so on, this trick would work fine. The issue is when you have a network where the different edges can have very different lengths. And that's certainly the case in many applications. Definitely road networks would be one where you have both sort of long highways and you have neighborhood streets. And potentially in financial transaction based networks you would also have a wide variance between the value of different transactions. And the problem then is some of these edge lengths might be really big. They might be 100, they might be 1,000. It's very hard to put operating bounds on how large these edge weights could be. So if you start wantonly replacing single edges with these really long paths of like 1,000, you've blown up the size of your graph way too much. So you do have a faithful representation of your old network, but it's too wasteful. So even though breadth-first search runs in linear time, it's now on this much larger graph. And we'd much prefer something which is linear time or almost linear time that works directly on the original graph. And that is exactly what Dijkstra's shortest-path algorithm is going to accomplish. Let's now move on to the pseudocode for Dijkstra's shortest path algorithm. So this is another one of those algorithms where no matter how many times I explain it, it's always just super fun to teach. And the main reason is because it exposes the beauty that pops up in good algorithm design. So the pseudocode, as you'll see in a second, is itself very elegant. We're just going to have one loop, and in each iteration of the loop we will compute the shortest path distance to one additional vertex. And by the end of the loop we'll compute shortest path distances to everybody. The proof of correctness, which we'll do in the next video, is a little bit subtle, but also quite natural, quite pretty. And then finally, Dijkstra's algorithm will give us our first opportunity to see the interplay between good algorithm design and good data structure design. So with a suitable application of the heap data structure, we'll be able to implement Dijkstra's algorithm so it runs blazingly fast, almost linear time, namely m times log n. But I'm getting little ahead of myself. Let me actually show you this pseudocode. At a high level, you really should think of Dijkstra's algorithm as being a close cousin of breadth-first search. And indeed, if all of the edge lengths are equal to one, Dijkstra's algorithm becomes breadth-first search. So this is sort of a slick generalization of breadth-first search when edges can have different lengths. So like our generic graph search procedures, we're going to start at the source vertex s, and in each iteration we're going to conquer one new vertex. And we'll do that once each iteration after m minus1 iteration, we'll be done. And in each iteration will correctly compute the shortest path distance to one new possible destination vertex v. So let me just start by initializing some notation. So capital X is going to denote the vertices that we've dealt with so far. And by dealt with, I mean we've correctly computed shortest path distance from the source vertex to every vertex in X. We're going to augment X by one new vertex in each iteration of the main loop. Remember that we're responsible for outputting n numbers, one for each vertex. We're not just computing one thing, we're computing the shortest path distance from the source vertex s to every other vertex. So I'm going to frame the output in terms of this array capital A. So for each vertex, we're going to have an entry in the array A, and the goal is at the end of the algorithm, A will be populated with the correct shortest path distances. Now to help you understand Dijkstra's algorithm, I'm going to do some additional bookkeeping which you would not do in a real implementation of Dijkstra's algorithm. Specifically, in addition to this array capital A in which we compute shortest path distances from the source vertex to every other destination, there's going to be an array capital B in which we'll keep track of the actual shortest path itself from the source vertex s to each destination v. So the arrays A and B will be indexed in the same way. There'll be one entry for each possible destination vertex v. Capital A will store just a number for each destination, shortest path distance. The array B in each position will store an actual path, so the shortest path from s to V. But again, you would not include this in an actual implementation. I just find in my experience it's easier for students to understand this algorithm if we think of the paths being carried along as well. So now that I've told you the semantics of these two arrays, I hope it's no surprise how we initialize them for the source vertex itself, s. The shortest path distance from s to itself is 0. The empty path gets you from s to s with length 0. There's no negative edges by assumption, so there's no way you can get from s back to s with a non-positive length, so this is definitely the shortest path distance for s. By the same reasoning, the shortest path from s to s is just the empty path, the path with no edges in it. So now let's proceed to the main while loop. So the plan is we want to grow this set capital X like a mold until it covers the entire graph. So in each iteration, it's going to grow and cover up one new vertex and that vertex will then be processed. And at the time of processing, we're responsible for computing the shortest path distance from s to this vertex and also figuring out what the actual shortest path from s to this vertex is. So in each iteration, we need to grow X by one node to ensure that we make progress. So the obvious question is, which node should we pick? Which one do we add to X next? So there's going to be two ideas here. The first one we've already seen in terms of all of these generic graph search procedures, which is we're going to look at the edges and the vertices which are on the frontier. So we're going to look at the vertices that are just one hop away from vertices we've already put into X. So that motivates that a given iteration of the while loop to look at the stuff we've already process, that's X, and the stuff we haven't already processed, that's v minus X. s, of course, starts in X and we never take anything out of X, so s is still there. In some generic iteration of the while loop, we might have some other vertices that are in X. And in a generic iteration of this while loop, there might be multiple vertices which are not in X. And now, as we've seen in our graph search procedures, there are general or edges crossing this cut. So there are edges which have one endpoint in each side, one endpoint in X and one endpoint outside of X. This is a directed graph so they can cross in two directions. They can cross from left to right or they can cross from right to left. So you might have some edges internal to X. Those are things we don't care about at this point. You might have edges which are internal to v- X. We also don't care about those, at least not quite yet. And then you got edges which can cross from X to v-X, as well as edges that can cross in the reverse direction, from v -X back to X. And the ones we're going to be interested in, just like when we did graph search and directed graphs, are the edges crossing from left to right, the edges whose tail is amongst the vertices we've already seen and whose head is some not yet explored vertex. So the first idea is that in each iteration of the while loop we scan, or we examine, all of the edges with tail in X and head outside of X. One of those is going to lead us to the vertex that we pick next. So that's the first idea, but now we need a second idea, because this is again quite underdetermined. There could be multiple such vertices which meet this criterion. So for example, in the cartoon in the bottom left part of this slide, you'll notice that there's one vertex here. Which is the head of an arc that crosses from left to right. And there's yet another vertex down here in v minus x, which again is the head of an arc which crosses from left to right. There are two options of which of those two to suck into our set x and we might want some guidance about which one to pick next. The key idea in Dijkstra is to give each vertex a score corresponding to how close that vertex seems to the source vertex s, and then to pick among all candidate vertices, the one that has the minimum score. Let me be more precise. Among all crossing edges, with tail on the left side and head on the right side, we pick the edge that minimizes the following criterion. The shortest path distance that we've previously computed from s to the vertex v, plus the length of the edge that connects v to w. This is quite an important expression, so I will call this Dijkstra's greedy criterion. This is a very good idea to use this method to choose which vertex to add to the set x, as we'll see. I need to give a name to this edge which minimizes this quantity over all crossing edges. Let's call it v star w star. For example, in the cartoon in the bottom left, maybe of the two edges crossing from left to right, maybe the top one is the one that has a smaller value of Dijkstra's greedy criterion. In that case, this would be the vertex v star and the other end of the edge would be the vertex w star. This edge, v star, w star is going to do wonders for us. It will both guide us to the vertex that we should add the x next, that's going to be w star. It's going to tell us how we should compute the shortest path distance to w star, as well as what the actual shortest path from s to w star is. Specifically, in this iteration of the wild loop, after we've chosen this edge v star w star, we add w star to x. Remember, by definition, w star was previously not in capitol X. We're making progress by adding it to x, that's one more vertex in x. Now x is supposed to represent all of the nodes that we've already processed. So an environ of this algorithm is that we've computed shortest path distances for everybody in x as well as the actual shortest paths. Now that we're putting w star in x, we're responsible for all this information, the shortest path information. What we're going to do is we're going to set the r estimate of w star's shortest path distance from s to be equal to the value of this Dijkstra's greedy criterion for this edge. That is, whatever our previously computed shortest path distance from s to v star was plus the length of the direct edge from v star to w star. Now a key point is to realize that this code does make sense. By which I mean, if you think about this quantity A(v), this is been previously computed. And that's because environ of this algorithm is we've always computed shortest path distances to everything that is in capital X. And of course, the same thing holds when we need to assign w star shortest path distance because v star was a member of capital X, we had already computed its shortest path distance. So we can just look up the v star entry position in the array a. Over in our picture on our left, we would just say, what did we compute the shortest path distance to v star previously? Maybe it's something like 17. And then we'd say, what is the length of this direct edge from v star to w star? Maybe that's 6. Then we would just add 17 and 6 and we would put 23 as our estimate of the shortest path distance from s to w star. We do something analogous with the shortest path itself in the array b. That is, again, we're responsible, since we just added w star to capital X, we're responsible for suggesting a path from s to w star in the b array. What we're going to do is we're just going to inherit the previously computed path to v star and we're just going to tack on the end one extra hop, namely the direct edge from v star to w star. That will give us a path from s all the way to w star via v star as an intermediate pit stop and that is the entirety of Dijkstra's Algorithm. I've explained all of the ingredients about how it works at a conceptual level. The two things I argue is, why is it correct? Why does it actually compute shortest paths directly to all of the different vertices, and then secondly, how fast can we implement it? The next two videos are going to answer both of those questions but before we do that, let's go through an example to get a better feel for how this algorithm actually works. I also want to go through a non example so that you can appreciate how it breaks down when there are negative edges, and that'll make it clear why do we need a proof of correctness because it's not correct without any assumptions about the edge lengths.

# 11-2-Dijkstra's Algorithm - Examples

So let's just see how it works in the same example we traced through earlier. So we start out just by initializing things in the obvious way, so the shortest path distance from s to itself we say is 0. And the shortest path from s to itself is just the empty path. And initially, our x is going to be just the source for text itself. So, now we enter the main while loop and so, remember in the while loop, we say well, let's scan all of the edges whose tail is in the vertices we've already looked at, whose tail is in x and whose head is outside of x. Now, in this first iteration, there are two such edges, there's the edge (s,v) and the edge (s,w). So how do we know which of these two to choose? Well, we evaluate Dijkstra's Greedy criterion. And so remember what that is, Dijkstra's Greedy score for a given edge (v,w) that's crossing the frontier is just the previously computed shortest path distance for the tail of the arc plus the length of the arc itself. So at this point, (s,v) has a greedy score of 0 + 1, which is 1, and the arc (s,w) has a greedy score of 0 + 4, which is 4. So obviously (s,v) is going to be the shorter of those two. So we use the edge (s,v), this is playing the role of v\*w\* on the previous slide. And the algorithm then suggests that we should add v to our set x. So we suck in v, and our new x consists of s, n, v, and it also tells us how to compute the shortest path distance and the shortest path from s to v. Namely, in the A array, we just write down what was the Dijkstra's greedy score for this particular edge, and that was 0 + 1, or 1. It also tells how to compute the shortest path for v. Namely, we just inherit the shortest path to the tail of the arc, which in this case, is the empty path from s to itself and then we tack on the end, we append the arc we used to get here, the arc S(v). So now we go to the next iteration of the while loop, so with our new set capital X consisting of s and v. And now again, we want to look at all edges which are crossing the frontier, edges that have tail in x and head outside x. And now we see there's three such crossing edges. There's (s,w), there's (v,w), and there's (v,t). All of those have the tail in x and the head outside of x. So we need to compute Dijksta's greedy score for each of those three and then pick the minimum. So let's go from bottom to top. So first of all, we can look at the arc (s,w). And the greedy score here is the shortest path distance for the tail, so it's 0 plus the length of the arc, which is 4. So here we get a 4 in this iteration. Then if we do this cross-bar edge, this (v,w) edge, the Dijkstra's greedy score is the A value, or the shortest path distance value of the tail. And we computed that last iteration, that A(V) value is 1, we add to that the length of the arc, which in this case is 2. So this edge (v,w) has a score of 3. Finally there's the arc (v,t) and here we're going to add 1, which is the shortest path distance of the tail of the arc plus the edge length, which is 6. So that has the worst score. So, since the edge (v,w) has the smallest score, that's the one that guides how we supplement x and how we compute the shortest path distances and the shortest path for the newly acquired vertex w. So the changes are, first of all, we enlarge x. So x is now everything, but t. And then how do we compute things for w? Well, the shortest path, so the r entry in the A array is just going to be Dijkstra's greedy score in the previous iteration. So that was 1+2, so it's going to be equal to 3. And then what is the shortest path? How do we fill up the array B? Well we inherit the shortest path to the tail of the arc, which in this case, is the arc (s,v) and then we append the arc that we used to choose this new vertex w, so that's the arc (v,w). So the new path is just the (s,v,w) path, okay, so it's what we compute is the shortest path from s to w in this graph. So now we proceed to the final iteration of Dijkstra's algorithm. We know what vertex we're going to bring in to x, it's going to be the vertex t, that's the only one left. But we still have to compute by which edge we discovered t and bring it in to the set x. So we have to compute the greedy score for each of the two crossing arcs, (v,t) and (w,t). And then this final iteration, the score for the arc (v,t) is unchanged. So this is still going to be the a value of its tail 1, plus the length of the arc, 6. So the score here is still 7, and now for the first time, (w,t) is a crossing edge of the frontier. And when we compute its score, it's the a value of its tail w, which is 3, plus the length of this arc which is 3, so I get a greedy score of 6. So by Dijkstra's greedy criterion, we picked the edge (w,t) instead of the edge (v,t), and of course, that doesn't matter who gets brought into X, but it does matter how we compute the A and B values for T. So in the final iteration, we compute (a,t) to be the Dijkstra's greedy score of the edge that we picked, which is the edge (w,t), and the score was 6. So we compute the shortest path distance from s to t to be 6. And then what is the path itself? Well, we inherit the shortest path to the tail of the arc that we used to discover t, so that's the shortest path to w, which we previously computed as being the path through v. And then we append the edge we used to discover t, so we append the edge (w,t). So the shortest path from s to t, we're going to compute as the zigzag path, s goes to v goes to w goes to t. And then now, dx is all the vertices, we've computed it for everything, this is our final output. The contents of the, especially the A array is the final output, shortest path distances from s to all of the four possible destinations. And if you go back and compare this to the example you went through the quiz, you will see at least on this example, indeed Dijkstra's algorithm corrects the shortest path distances. Now I've said it before, I'm going to say it again. Someone shows you their algorithm works just on some examples, especially a pretty simple four note example, you should not jump to the conclusion that this algorithm always works. Sometimes algorithms work fine on small examples, but break down once you go to more interesting complicated examples. So I definitely owe you a proof. The Dijkstra's algorithm works not only in this network, but in any network. And actually it doesn't work in any network, it's only going to work in any network with non-negative edge lengths. So to help you appreciate that, let's conclude this video with a non example, showing what goes wrong in Dijkstra's algorithm when you have networks with negative edge lengths. So before I actually give you a real non-example, let me just answer a preliminary question, which you might have, and this would be a very good question if it's something that has occurred to you. The question would be well, why are these negative edge links such a big deal? Why can't we just reduce shortest path computation with negative edge links to the problem of computing shortest paths with non-negative edge links, right? So why don't we just sort of clear things out? We just add a big number to all the edges, that makes them all non-negative and then we just run Dijkstra's algorithm and we're good to go. So this is exactly the sort of question you should be looking to ask if as a computer scientist, as a serious programmer. When confronted with a problem, you always want to look for ways to reduce it to simpler problems that you already know how to solve. And this is a very natural idea of how to reduce a seemingly harder shortest path problem to one we already know how to solve using Dijkstra's algorithm. The only problem is, it doesn't quite work. Why doesn't it work? Well, let's say you have a graph, and the most negative edge is minus ten. So, all the other edge links are minus 10 and above. So then, what you want to do is add 10 to every single edge in the network, and that ensures that all the lengths are non-negative. Run Dijkstra's algorithm, get your shortest path. The issue is that different paths between a common origin and destination have differing numbers of edges. So, some might have five edges, some might have two edges. Now, if you add 10 to every single edge in the graph, you're going to change path lengths by different amounts. If a path has five edges, it's going to go up by 50 when you add 10 to every edge. If a path has only two edges, it's only going to go up by 20 when you add 10 to every edge. So as soon as you start changing the path lengths of different paths by different amounts, you might actually screw up which path is the shortest. The path which is shortest of the new edge lengths need not be the one the shortest under the old edge lengths. So that's why this reduction doesn't work. To be concrete, let's look at this very simple three vertex graph where vertices s, v, and t and edge lengths as shown 1, -5 and -2. Now what I hope is clear is that in this graph, the shortest path, the one with the minimum length, is the too hot path, s, v, t. That has length minus 4. The direct s,t arc has length minus 2, which is bigger than minus 4. So the upper path is the shortest path. Now suppose we try to massage this by adding a constant to every edge so that all edge links were non-negative. We have to add 5 to every edge, because that's the biggest negative number, the (v,t) edge. So that would give us new edge lengths of 6, and 0, and 3. And now the problem is, we have changed which path is the shortest one. We added 10 to the top half and only 5 to the bottom half and as a result, they've reversed. So now the bottom path (s,t) is actually the shorter one, so if you run Dijkstra's on this graph, it's going to come back with a path (s,t), even though that's not in fact the shortest path in the original network, the one that we actually care about, okay. So that's why you can't just naively reduce shortest paths with negative edge lengths to shortest paths with non-negative edge lengths. Moreover, on this very same super simple three nug graph, we can try running Dijkstra's shortest path algorithm. It's perfectly well defined, it'll produce some output, but it's actually going to be wrong. It is not going to compute shortest path distances correctly in this graph, so let me show you why. Of course, the initialization will work as it always does, so it's going to start by saying the shortest path distance from s to itself is 0 by the empty path. And then, what's it going to do next? It's going to say, okay, well, we need to enlarge the set capital, x, by one vertex, and there are two crossing edges as the (x,v) edge and the (s,t) edge. And what's it going to do? It's going to use the Dijkstra's greedy score. So, the score of this upper edge is going to be 1, and the score of this bottom edge is going to be minus 2. because remember, you take the previously computed shortest path value of the tail, that is 0 in both cases, and then you add the edge length. So the edge lengths are 1 and minus 2, so the scores are 1 and minus 2. Which of these is smaller? Well evidently, the (s,t) arc has the smaller score, minus 2. So what is Dijkstra's algorithm going to do? It's going to say, yes, let's go for this edge, (s,t). Let's bring T into the set capital X. T is now part of the conquered territory. And of course, as soon as you bring a node into the set X, into conquered territory, you have to commit or Dijkstra's algorithm chooses to commit to its shortest path distance and its shortest path. What is the definition of its shortest path distance, as computed by Dijkstra? Well it's just a greedy score. So it's going to assign the vertex t, the shortest path distance of minus 2, and the path is going to be just the arc, (s,t). But notice that this is in fact wrong. The shortest path distance from s to t is not minus 2 in this graph. There's another path, namely the one that goes through v, that has length minus 4, less than minus 2. So Dijkstra computes incorrect shortest path distances on this trivial three note graph. So to summarize the story so far, we've described Dijkstra's algorithm. I've showed you that it works in a very simple example that doesn't have negative edge lengths. And I've showed you that it doesn't work in an even simpler example that does have negative edge lengths. So I've both given you some plausibility that it might work generally, at least for non negative edge lengths, but I've also tried to sow some seeds of doubt. That it's not at all clear at this point if Dijkstra's algorithm is always correct or not, even if you have non negative edge lengths. And certainly, if it is always correct, there better be a foolproof argument for why. You should be demanding an explanation of a claim that Dijkstra is correct in any kind of generality. That's the subject of the next video.

# 11-3-Correctness of Dijkstra's Algorithm

In this video, I'll prove to you that Dijkstra's algorithm does indeed compute to correct shortest paths in any directed graph where all edge links are non negative. So let me remind you about what is Dijkstra's algorithm, it's very much in the spirit of our graph search primitives, in particular breath first search. So there's going to be a subset x of vertices, which are the ones that have been processed so far. Initially x contains only the source vertex. Of course the distance from the source vertex to itself is 0, and the shortest path from s to itself is the empty path. So then we'll have a main while loop, that's going to be n-1 iteration, and each iteration will bring one vertex which is not currently in x into capital X. And a variant that we are going to maintain, is that all the vertices in x we will have computed estimates of the shortest path distance from x to that vertex and also we'll have computed the shortest path itself from x to that vertex. Remember our standing assumption stated in the previous video, we're always going to assume there's at least one path from the source vertex s to every other destination v. Our job is just to compute the shortest one. And also, we have to assume that the edge links are non negative as we've seen otherwise Dijkstra's algorithm might fail. Now, the key idea in Dijkstra's algorithm is a very careful choice of which vertex to bring from outside of x into capital X. So what we do is we scan the edges crossing the frontier. Meaning given the current edges vertices that we've already processed, we look at all of the edges whose tail has been processed and whose head has not been processed. So the tails in capital X, the head is outside of x that is, they cross the cut from left to right in the diagrams that we usually draw. Now, there may be many such edges. How do we decide amongst them? Well, we compute the Dijkstra's greedy score for each. The Dijkstra greedy score is defined as the shortest path distance we computed for the tail and that's been previously computed because the tail's in capital X. And then we add to that the length contributed by this edge itself by the edge vw which is crossing the cut from left to right. So amongst all edges crossing the cut from left to right we compute all those Dijkstra greedy scores we pick the edge with the smallest greedy score Calling that edge just v\* w\*. For the purposes of notation, w\* is the one that gets added to x. So it's the head of the arc for the smallest three score, and then we compute the shortest path distance of that new vertex w\* to be the shortest path distance to v\* plus the length contributed by this edge v\* w\* and then was the shortest path. It's just the shortest path previously computed to v star plus this extra edge v\* w\* tacked onto the end. Here's the formal statement we're going to prove. For this video, we're not going to worry at all about running time, that'll be the discussion of the next video. We'll discuss both the running time of the basic algorithm and a super fast implementation that uses the heat data structure. For now, we're going to just focus on correctness. So the claim is that for every directed graph, not just the four node, five arc example we studied. As long as there's no negative edge links, Dijkstra's algorithm works perfectly. It computes all the correct shortest path distances. So just to remind you about the notation, what does it mean to correct all shortest path distances correctly? It means that what the algorithm actually computes, which is a(v), is exactly the correct shortest path distance, which we were denoting by L(v) in the previous video. Both the algorithm and the proof of correctness where established by Esther Dijkstra this was back in the late 1950s. Dijkstra was a Dutch computer scientist, and certainly one of the forefathers of the field as a science, as an intellectual discipline. He was awarded the ACM Turing award, so that is the Nobel Prize in computer science effectively. I believe it was 1972, and he worked a long time in the Netherlands, but then also spent a lot of his later career at UT Austin. So the way this proof is going to go is going to be by induction. And basically, what we're going to do is we're going to say every iteration, when we have to commit to shortest path distance to some new vertex, we do it correctly. And so then the form of the induction will be, well given that we made all of our previous decisions correctly, we computed all our earlier shortest paths in the correct way. That remains true for the current iteration. So formally, it's induction on the number of iterations of Dijkstra's algorithm. And as is more often than not the case in proofs by inductions the base case is trivial. So that just says before we start the y loop, what do we do? Well we commit to the shortest path distance from s to itself. We set it to 0, we set the shortest path to be the empty path, that is of course true. Of course, even here we're using the fact that there are no edges with negative edge length. That makes it obvious that sort of having a non empty path can get you negative edge length better than 0. So the first choice path computation we do s to s is trivially correct. The hard part of course is the inductive step justifying all of the future decisions done by the algorithm. And of course, mindful of that example that not example we had at the end of the previous video in the proof by induction, we'd better make use of the hypothesis that every edge has non negative length. Otherwise the theorem would be false. So we better somewhere in the proof use the fact that edges cannot be negative. So let's move on to the inductive step. Remember in the inductive step, the first thing to do is state the inductive hypothesis. You're assuming you haven't made any mistakes up to this point. Let's be a little bit more formal about that. So that is everything we computed in the past. What did we compute in the past? Well for each vertex which is in our set capital X for each vertex that we've already processed, we want to claim that our computed shortest path distance matches up exactly with the true correct shortest path distance. So in our running notation, for every already processed vertex, so for all vertices v, in our set capital X. What we computed as our estimate of the shortest path distance for v is in fact, the real shortest path distance. And also, the computed shortest path is in fact, a true shortest path from s to v. So again remember, this is a proof by induction. We are assuming this is true, and we're going to certainly make use of this assumption when we establish the correctness of the new iteration, the current iteration. What happens in an iteration? Well, we pick an edge which we've been calling v\* w\*. And we add the head of sets w\* to the set X. So let's get our bearings and remember what Dijkstra's algorithm computes as the shortest path and shortest bath distance for this new vertex w\*. So by the definition of the algorithm we assign as a shortest path from s to w\*. The previously computed purportedly shortest path from s to v\*, and then we tack on the end the direct arc, v\* w\*. So pictorially, we already had some path that started at s and ended up at v\*, and then we tack on the ends this arc going to w\* in one hop. And this whole shebang is what we're going to assign as B of w\*. So let's use the inductive hypothesis. Inductive hypothesis says that all previous iterations are correct. So that is any shortest path we've computed in a previous iteration is in fact a bona fide shortest path from the source x to the vertex. Now v\*, remember is in x. So that was previously computed. So by the inductive hypothesis, this path bv\*, from s to v\*, is in fact a true shortest path from s to v\* in the graph. So therefore, it has length l of v\*. Remember, l of v\* is just by definition the true shortest path distance in the graph from s to v\*. Now, given that the path that we've exhibited s to w\*, is just the same one as we inherited the v\* plus this extra edge tacked on. It's pretty obvious what the length of the left hand side is. It has length, just the length of the old path which we just argued is the shortest path distance from sw\* plus the length of this arc that we tacked on. That's going to be lv\* w\*. So by the definition of the algorithm, what we compute for w\* is just the Dijkstra's greedy score which is just the computer choice path distance to the tail. The v\* plus the length of the direct edge. By the inductive hypothesis, we've correctly computed all previous choice path distances. V\* is something we computed in the past by inductive hypothesis is correct. So this is equal to l of v\* by the inductive hypothesis. So don't worry if you're feeling a little lost at this point. We've actually really done no content in this proof yet. We haven't done the interesting part of the argument. All we've been doing is setting up our dominoes, getting them ready to be knocked down. So what have we done in the current iteration? Well first of all, our estimate of the shortest path distance from the source to w\*, to the new vertex that we're including in the set capital X, is the true shortest path distance to v\* plus the length of the edge from v\* to w\*, that's the first thing. Secondly, the path that we have in the v array is a bona fide path from s to w\* with exactly this distance. And the point is, now it's clear what has to be proven for us to complete the inductive step and therefore, the proof of correctness of Dijkstra's algorithm. So what do we need to proof? We need to proof that this isn't just any old path that we've exhibited from s to this vertex w\*, but that it's the shortest path of them all. But differently we need to show that every other sw\* pattern in this graph has length at least this circled value. So let's proceed let's show that no matter how you get from the source for test to this destination w\*. The total length of the path you travel is going to be at least this circled value, at least L(v\*) + lv\*w\*. Now on the one hand, we don't have a lot going for us, because this path P could be almost anything. It could be a crazy looking path. So how do we argue that it has to be long? Well, here's the one thing we've got going for us for any path P that starts in s and goes to w\*. Any such path must cross the frontier. Remember, it starts on the left side of the frontier, it starts at the source vertex, which is initially and forever in the set capital X. And remember that we only choose edges that cross the frontier whose head is outside of x. And w\* is exactly the head of the edge we chose in this iteration, so this is not an x. So any path that starts in x and goes outside of x at some point it crosses from one to the other. So let's think about the graph and it's two pieces, that it's the left of the front tier and not to the right. The stuff is already processed and the stuff which is not been processed. S of course, is on the left hand side, and at the beginning of this iteration of the while loop, w\* was on the right hand side. Any path, no matter how wacky has to at some point, cross this frontier. Maybe it does it a bunch of times, who knows but it's gotta do it once. Let's focus on the first time it crosses the frontier, and let's say that it crosses the front here with the vertex y going to the vertex z. That is any path P has the form where there's an initial prefix,where are the vertices are in x. And then there's some first point at which it crosses the frontier and goes to a vertex which is not an x, we're calling the first such vertex outside of x, z. And then it can skip back and forth who knows, but certainly it ends up in this vertex w\* which is not an x. So we're going to make use of just this minimal information about an arbitrary path P. And yet this will give us enough of a foothold to lower bound its length. And this lower bound to be strong enough, we conclude that our path that we computed is the best, smaller than any possible competitor. So let's just summarize where we left on the previous slide. We established that every directed path from s to w\* p, no matter what it is has to have a prescribed form, where it ambles for a while inside x and then the portal through which it escapes x for the first time we're calling y. And then the first vertex it sees outside of x is z and there has to be one. And then it perhaps ambles further and eventually reaches w\*. It could well be that z and w\* are exactly the same, that's totally fine for this argument. So here's one of our competitors, this path p and I have to show it's at least as long as our path. So we need a lower bound on the length of this arbitrary path from s to w\*. So let's get that lower bound by arguing about each piece separately, and then invoking Dijkstra's greedy criterion. So remember, we said we better use the hypothesis that edge links are non negative, otherwise we're toast, otherwise we know the algorithm is not correct. So here's where we use it. This final part of the path from z to w\*, if it's not empty then it's gotta have non negative length right. Every edge as part of this subpath has non negative edge length, so the total length of this part of the path is non negative. So y to z by construction is direct arc. Remember, this is the first arc that path p uses to go from x to get outside of x. So that's how it escapes, the conquer territory x and this just has some length, l of yz. So that leaves the first part of this path, the prefix of this path that lies entirely in capital X. So how do we get a lower bound in the length of this path? Well, let's begin with something trivial. This is some path from s to y, so certainly it's as least as long as a shortest path from s to y. And now, we're going to use the inductive hypothesis again. So this vertex y, this is something we treated in a previous iteration. This belongs to the set capital X, we've already processed it, we've already computed our estimate of it's shortest path length, and the inductive hypothesis assures us that we did it correctly. So whatever value we have hanging out in our array capital A, that is indeed the length of the true shortest path. So the length of the shortest sy path is l(y) by definition, and it's A(y) by the inductive hypothesis, and now we're in business. So what does this mean we can say about the total length of this arbitrary path P? Well, we've broken it into three pieces and we have a lower bound on the length for each of the three pieces. Our lower bounds are, our computed shortest path distance to y, the length of the direct edge from y to z and 0. So adding those up, we get that the length of path P is at least our computed shortest path distance to y plus the length of the arc from y to z. So why is this useful? Well, we've got one remaining trick up our sleeve. There's a hypothesis which is presumably very important, which we have not yet invoked. And that is the choice of Dijkstra's greedy criterion at no point in the proof yet have we used the facts that we select which vertex to add next according to Dykstra's greedy score. So that is going to be the final nail in the coffin, that's what's going to complete the proof. How do we do that? Well we have taken an arbitrary path P, we have lower abounded it's length, in terms of the computed shortest path distance up to the last vertex of this prefix y plus the arc length to get from x to l set of X, zyz. So remember, this means y is on the left part of the frontier and z is not. And therefore in this iteration, the edge yz was totally a candidate for us to use to enlarge our frontier. Remember, we looked at all of the edges crossing from left to right. Yz is one such edge and amongst all of them, we chose the one with the smallest Dijkstra's greedy score. That was the Dijkstra's greedy criterion. So what have we shown? We've shown that the length of our path is no more than what's a lower bound on the length of this arbitrary other path P. So this completes the proof. So let me just remind you of all the ingredients, in case you got lost along the way. So what we started out with is we realized our algorithm or Dijkstra's algorithm it does compute some path from s to w\*. It just takes the path it computed previously to v\*, and it just depends this final hop at the end. So that gives us some path from s to w\* moreover, it was easy to figure out exactly what the length of that path is. And the length of the path that we came up with is exactly the circled quantity at the bottom of the slot. It's the shortest path distance from s to v\* plus the length of the direct arc from v\* to w\*. So that was how well we did. But we had to ask the question, is it possible to do better? We're trying to argue that our algorithm does the best possible, that no competing path could possibly be shorter than ours. So how did we do that? Well, we considered an arbitrary competing path P. The only thing we know about it is that it starts at s and it ends up at w\*. And we observe that any path can be decomposed into three pieces. A prefix, a direct edge, and a suffix. Then we give a lower bound on this path P. The direct edge, the length is just whatever it is. The suffix, we just use the trivial lower bound that's at least 0. And that's where we use the hypothesis that every edge has non negative edge length. And for the prefix, because that's all in the stuff we already computed, we can vote the inductive hypothesis and say, well whatever this path is, it goes from s to come vertex and y. So at least the shortest path distance from s to y which is something we computed in a previous iteration. We lower bounded the length of any other path in terms of the Dijkstra's greedy score for that path. Since we choose the path with the best greedy score, that's why we wind up with the shortest path of them all, from s to w\*. This of course, is embedded in an outer proof by induction on the number of iterations, but this is the inductive step, which justifies a single iteration. Since we can justify every iteration giving correctness to the previous ones. That means by induction, all of them are correct. So all of the shortest paths are correct. And that is why Dijkstra's algorithm correctly computes shortest paths and any directed graph with non negative edge lengths.

# 11-4-Dijkstra's Algorithm - Implementation and Running Time

In this video we'll discuss how we actually implement Dijkstra's shortest path algorithm. And in particular, using the heap data structure, we'll give a blazingly fast implementation, almost linear time. Let me just briefly remind you of the problem we're solving. It's the single source, shortest path problem. So we're given the directed graph and a source vertex, S. We're assuming that there's a path from S to every other vertex, V. If that's not true, we can detect it with an easy pre-processing step, so our task then is just to find the shortest path amongst all of them from the source vertex, S to each possible destination, V. Moreover, every edge of the graph has a non-negative edge length which we're denoting, else of V. So recall that Dijkstra's algorithm is driven by a single Y loop. So we're going to add one additional vertex to an evolving set capital X as the algorithm proceeds. So X is the vertices that have been processed so far. We maintain the invariant that for every processed vertex we've computed what we think the shortest path distance is to that vertex. So initially X is just the source vertex S. Of course, the shortest path distance from S to itself is zero. And then the cleverness of Dijkstra's algorithm is in how we figure out which vertex to add to the set capital X each iteration. So the first thing we do is we. Focus only on edges that cross the frontier edges that have their tail in capital X and their head outside of capital X. Now, of course, there may be many such edges, Edges that cross this frontier and we use Dijkstra's Grady criterion to select one of them. So for each crossing edge, each edge with a tail and X and head outside X, we compute the Dijkstra Grady score that is defined as the previously computed shortest path distance to the tail of the arc plus the length of the arc. So we compute that for each crossing edge and then the minimum edge we're calling it V star W star. That determines how we proceed. So we add the head of that arc W star to the set capital X and then w e compute the shortest path distance to W star to give the previous. See computed shortest fast distance to V Star plus the length of this extra [inaudible] V Star, W Star. Now back when I explained this algorithm I did it using two arrays, array capital A and array capital BA Is what computed the shortest path distances, and remember that's what the problem asks us to compute. And for clarity I also filled up this array capital B just to keep track of the shortest paths themselves. Now if you look at the code of this algorithm, we don't actually need the array capital B for anything. When we fill in the array capital A, we don't actually refer to the B array. And so now that we're gonna talk about real implementations of Dijkstra; I'm actually gonna cross out all of the instructions that correspond to the B array. Okay? Because you would not, as I told you earlier, use this in a real implementation of Dijkstra. You would just fill in the shortest path distances themselves. So in the next quiz, what I want you to think about is the running time of this algorithm if we implemented it more or less as is, according to the pseudo code on this slide without any special data structures. And in the answers to the quiz, we're going to be using the usual notation where M denotes the number of edges in the graph, and N denotes the number of vertices of the graph. So the correct answer to this quiz is the fourth one that the straightforward implementation of Dijkstra's algorithm would give you a running time proportional to the product of the number of edges and the number of vertices. And the way to see that is to just look at the main while loop and look at how many times it executes and then how much work we do per iteration of the while loop if we implemented it in a straightforward way. So there's gonna be N minus one iterations of the while loop. And the reason is, is that the algorithm terminates once every single vertex has been added to capital X. Their end vertices, initially there's one vertex in X. So after N m inus one iterations, we'll have sucked up all of the vertices. Now what's the work done in each wild loop? Well basically, we do naively a linear scan through all of the edges. We go through the edges. We check if it's an eligible edge, that is if its tail is in X and its head is outside of X. We can keep track of that just by having an auxiliary bullion variable for each vertex. Remembering whether it's an X or not. And then amongst all of the illegible edges, those crossing the frontier, we just, by exhaustive. Search remember which edge has the smallest Dijkstra store- score, now we can compute the Dijkstra score in constant time for each of the edges. So that's a reasonable algorithm. We might be able to get away with graphs that have say hundreds or thousands of vertices using the straight forward of implementation, but of course, we'd like to do better. We'd like the algorithm to scale up to much larger graphs, even graphs with potentially say a million vertices So the answer is yes, we can do better. Not by changing the algorithm, but, rather, changing how we organize the data as the algorithm proceeds. So this will be the first time in the course where we use a data structure to get an algorithmic speed-up. So we're gonna see a really lovely interplay between, on the one hand, algorithm design and, on the other hand, data structure design in this implementation of Dijkstra's algorithm. So you might well ask what's the clue that indicates that a data structure might be useful in speeding up Dijkstra's shortest path algorithm. And the way you'd figure this out is you'd say, well, where is all this work coming from? Why are we doing a linear amount of work in the edges for a linear number in the vertices iterations? Well, at each iteration of this while loop, what we're doing is, we're just doing an exhaustive search to compute a minimum. We look at every edge, we look at those that cross the frontier, and we compute the one with the minimum Dijkstra score. So we could ask ourselves, oh, if we're doing minimum comp utations over and over and over again, Is there some data structure which, whose raison d??tre, whose reason for being is in fact to perform fast minimum computations? And in fact there is such a data structure. It's the heap data structure. So in the following description of a fast implementation of Dijkstra's algorithm, I'm going to assume you're familiar with this heap data structure. For example, that you watched the review video elsewhere on the course site that explains it. So let me just remind you with a lightning-quick review of what we learned in that video, So heaps are generally logically thought of as a complete binary tree, even though they are usually implemented as a laid-out linear array. And the key property that you get to leverage but that you also have to maintain in a heap is the heap property that at every node the key at that node has to be at least as small as that of both of the children. This property ensures that the smallest key of them all has to be at the root of this tree. To implement extract menu, just pluck off the roots. That's what you return. That's the minimum element. And then you swap up the, bottommost rightmost leaf. The last element, Make that the new root, And then you bubble that down as necessary to restore the heap property. When you do insertion, you just make the new element the new last leaf, bottommost rightmost leaf, and then you swap up as needed to restore the heap property. When we use heaps in Dijkstra's algorithm we're also going to need the ability to delete an element from the middle of the heap. But again you can do that just by swapping things and doubling up or down as needed. I'll leave it as an exercise for you to think through carefully, how to delete elements from the middle of a heap. Because you're maintaining the heap as an essentially perfectly balanced binary tree, the height of the tree is roughly the log base two of N, where N is the number of elements in the heap. And because for every operation, you implement it just by doing a constant amo unt work at each level of the tree, all of these operations run in O of log N time, where N is the number of items that are being stored in the heap. As far as the intuitive connection between the heap data structure and Dijkstra's Algorithm. In the main wild loop of Dijkstra's Algorithm, we're responsible for finding a minimum, every single iteration. What are heaps good for? They're good for finding minimums in logarithmic time. That sounds a lot better than the linear time we're spending in the naive implementation of Dijkstra's Algorithm. So let's now see how to use heaps to speed up Dijkstra's shortest path algorithm. Now because every iteration of the wild loop is responsible for picking an edge, you might expect that we're going to store edges in the heap. So the first subtle but really good idea is to actually use a heap to store vertices rather than edges. Going back to the pseudo-code [inaudible] algorithm, remember that the only reason that we focused on an edge. Well so that we can then deduce which vertex, namely the head of that edge, to add to our set capital X. So we're just going to cut to the chase, we're just going to keep vertices not yet in X and then when we extract them in from the heap, it'll tell us which is the next vertex to add into the set capital X. So the picture we're going to wanna have in mind is Dijkstra's choice path algorithm at some intermediate iteration. So there'll be a bunch of vertices in the, the set capital X source vertex plus a bunch of other stuff that we've sucked into the set so far. And then there'll be all the vertices we haven't processed yet. A big group V minus X. Then there's gonna be edges crossing this cut in both directions from X to V minus X and vice versa. Now before I explain the second invariant, let's just recall what the straightforward implementation of Dijkstra's Algorithm needs to do. What it would do is search through all the edges and it would look for any eligible edges. Those with tail and X, and head and V minus X. So in this picture, there w ould be three such edges. I've drawn the example so that two of the edges, the top two edges, both share a common head vertex whereas the third edge has its own head vertex. The straightforward of limitation of Dysktra's Algorithm we?d compute Dystra's greedy score for each of these three edges And remember, by definition, that's the previously computed shortest path distance to the tail of the arc V, plus the length of the arc VW. So the straightforward implementation just computes this. In this case, it would compute it for three edges. And whichever the three edges won had the smallest score. The head of that edge would be the next vertex that gets added to X. So let me specify the second invariant, and then I'll tell you how to think about it. So, because we're storing vertices rather than edges in the heap, we're going to have to be fairly clever with the way we define the key of a vertex that's in this heap. [sound] So we're going to maintain the property that the key of a vertex V is the smallest greedy Dijkstra score of any ver, any edge which has that vertex as its head. So let me show you what I mean in terms of our example, where we have three crossing edges. Suppose for these three edges in the upper right that happen to have of Dijkstra [inaudible] scores of seven, three, and five. Let's look at what the key should be for each of these three vertices I've drawn in V minus X. Now for the timeline vertex this is pretty interesting. There are two different edges whose tail is in X, and have this vertex as their head. So what should the key of this vertex be? Well, it should be the smallest Dijkstra greedy score of any of the edges whose tail lies on the left-hand side that terminate at this vertex. So there's two candidate edges. One has Dijkstra greedy score three. One has Dijkstra greedy score seven. So the key value should be three, the smaller of those two. Now, the second vertex, there's only a single edge that has tail in X and that terminates at this vertex. So the key for this vertex should jus t be the score at that weak edge. So in this case that's gonna be five. And then this poor third vertex, there's actually no edges at all, that, started X and terminated at this vertex. There's only one arc going the wrong direction. So for any edge, sorry, for any vertex outside of X that doesn't have any eligible edges terminating at it, we think of the key as being plus infinity. So the way I recommend thinking about these heap keys is that we've taken what used to be one round tournament, winner takes all And we've turned it into a two round knockout tournament. So in our straightforward implementation of Dijkstra's algorithm, we did a single linear search through all the edges, and we just computed the [inaudible] Dijkstra's score for each and we picked the best. So in this example we would have discovered these three edges in some order. Their scores are three, five, and seven. And we would have remembered the edge with score three as being the best. That would have been our winner of this winner take all tournament. Now when we use the heap, it, we're factoring it into two rounds. So first, each vertex in V minus X runs a local tournament. To elect a local winner, so each of these vertices in V minus X. Says, well let me look at all of the edges. For whom I'm the head and also the tail of that edge is in X. And amongst all of those edges that start in X and terminate at me, I'm going to remember the best of those. So that's the winners of the local tournament of the first round. And now the heap is only going to remember this set of first round winners. Right, there's no point in remembering the existence of edges who aren't even the smallest score that terminate at a given vertex, because we only care about the smallest score overall. Now when you extract min from the heap, that's in effect. Executing the second and final round of this knockout tournament. So each of the vertices of V minus X has proposed their local winner. And then the heat in an extract min just chooses the best of all of those local winners. So that's the final proposed vertex that comes out of the heap. So the point is that if we can successfully maintain these two invariants, then, when we extract min from this heap, we'll get exactly the correct vertex, W star, that we're supposed to add to the set capital X next. That is, the heap will just hand to us on a silver platter exactly the same choice of vertex that our previous exhaustive search through the edges would've computed. The exhaustive search was just computing the minimum in a brute force way, in a single winner take all tournament. The heap implemented in this way chooses exactly the same winner. It just does it in this 2-round process. Now, in Dijkstra's algorithm, we weren't supposed to merely just find the vertex W star to add to X. We also had to compute its shortest path distance. But remember, we computed the shortest path distance as simply the Dijkstra greedy score. And here the Dijkstra greedy score is just going to be the key for this heap that's immediate from invariant number two. So we're using the fact here that our keys are, by definition, just. The smallest greedy scores are edges that stick into that vertex W STAR so again exactly replicating. The computation that we would have done in the straightforward implementation, just in a much slicker way. Okay? But we're adding exactly the same vortices, in exactly the same order, and we're computing exactly the same shortest path distances in this heap of notation, provided of course that we do successfully maintain these two invariants throughout the course of the algorithm. So that is now what I owe you. We have to pay the piper. We've shown that if we can have a data structure with these properties. Then we can simulate the straight forward implementation now I have to show you how we maintain these invariants without doing too much work. All right. So maintaining invariant number one will really take care of itself. Really sort of by definition the vertices which remain in the heap are those that we haven't process ed yet, and those are the ones that are outside of capital X. So really the trick is, how do we maintain invariant number two? Now before I explain this let me point out, that this is a tricky problem. There is something subtle going on. So as usual, I want you to think about this shortest path algorithm at some intermediate iteration. Okay? So take a, take a snapshot. A bunch of vortices have already been added to X. A bunch of vortices are still hanging out in the heap. They haven't been added to X. There's some frontier, there's a, just crossing, possibly in both directions. And suppose at the end of a current iteration we identify the vortex W, which we're going to extract from the heap and conceptually add to the set X. Now the reason things complicated is when we move a vortex from outside X to inside X. The frontier between X and V minus X changes. So in this picture, the old black X becomes this new blue X. And what's really interesting about the frontier changing is that then the edges which cross the frontier change. Now, there might be, there are some edges which used to cross the frontier and now don't. Those are the ones that are coming into W. Those we're not so concerned with. Those don't really play any role. What makes things tricky is that there are edges which used to not be crossing the frontier but now they are crossing the frontier. And those are precisely the edges sticking out of W. So in this picture there are three such edges which I will highlight here in pink. To see why it's tricky when new edges all the sudden are crossing a frontier let's remember what invariant number two says. It says that for every vertex which is still in the heap, which is not yet in X, the key for that vertex better be The smallest Dijkstra Grady score of any edge which comes from capital X and sticks into this vertex of V. Now in moving one vertex into X, namely this vertex W, now there can be new edges sticking into vertices which were still on the heap. As a result, the appropriate key value for vertices i n the heap might be smaller. Now the W has been moved into X. And the candidates for the vertices in the heap whose keys might have dropped are precisely those vertices on the other end of edges sticking out of W. So summarizing, the fact that we'd added a new vertex to capital X and extracting something from the heap, it's potentially increased the number of crossing edges across the frontier, because the frontier has changed. And therefore, for vertices that remain in the heap, the smallest greedy score of an edge that sticks into them from the set X might have dropped. So we need to update those keys to maintain invariant number two. Now, that's the hard part. Here's what we have going for us. We've damaged the keys perhaps by changing the frontier, but the damage is local. We can understand exactly whose keys might have dropped, so as suggested by the picture, the vertices whose keys we need to update are precisely those at the head of edges that stick out of W. So for each outgoing edge from W, the vertex we just extracted from the heap, we need to go to the other end of the edge and check if that vertex needs its key to be decreased. So here's the pseudo code to do this. So when we extract the vertex W from the heap, that is when we conceptually add a new vertex W. To the set X, thereby changing the frontier, we say, well, you know, we know the only vertices that might have to have their key changed, they're the ones on the other side of these outgoing arcs from W. So we just have a simple iteration over the outgoing edges, W V, from the vertex V. Now I haven't shown you any edges in the picture like this, but there might well be some edges where the head of the arc V is also in the set X, is also already been processes. But anything in X is not in the heap. Remember, the heap is only the stuff outside of X. So we could care less about the stuff outside. Of the heat, for not maintaining their keys. So we do an extra check. If the head of this edge is in fact still in the heap, that is if it's not in X So i n the picture, for example, this would be true for all three of the vertices that are on the other end of arcs pointing out of W. And for each of these vertices V, we update its key. And the way we're going to update its key is, we're just going to rip this vertex out of the heap. We're going to recompute its key and constant time, and then we're going to reinsert it into the heap. And since all heap operations take logarithmic time, this key update will be logarithmic time. As an additional optimization, I wanna point out that if one of these vertices V's key does change, it can only change in one way. So remember, what is the key? The key is the smallest Grady Dijkstra score of all of the edges that start next and stick into this vertex. So that's the local tournament or the first round tournament happening at this vertex V. Now the only thing which has changed. Before and after we added this vertex, W to X, is that now one new edge is sticking into this vertex, V. All of the old edges sticking into it from X are still sticking into it, and now there's one extra candidate in its local tournament, namely this edge, WV. So either WV is the local winner; either it has the smallest Dyxtra-Greedy score of them all. That terminated this vertex, or it doesn't, in which case the previous winner is still the new winner. So if that is, the new key value can only be one of two things. Either it's the old key value--that's the case where this. Extra entrance, the edge from W to V is irrelevant. Or, if it's changed, it has to have changed to the [inaudible] score of this edge, W-V. And the formula for that is the shortest path distance. That we just computed for W where W has been processed at this point plus the link of the direct arch from W M V. And again conceptually this formula is just a greedy Dijkstra score for the arc WV. The new entrance in V's local first round tournament. So now, having updated V's key appropriately, so that invariant #two is restored. And once again, the key of every vertex does reflect the sma llest greedy, Dijkstra greedy score of any edge sticking into it from the set X. We can safely reinsert this node back into the heap with its new key value. And these three lines together are just a key update in logarithmic time, for one of these vertices that's at the other end of an arc sticking out of the vertex W. So let's tally up the running time in this new implementation. One thing I want you to check, and this will definitely help you understand this refined implementation of Dijkstra's algorithm, is that essentially all the work done is through the heap API. That is, all of the running time that we have to account for is in heap operations. We don't really do nontrivial work outside of heap operations. And again recall that the running time of any heap operation is logarithmic in the number of elements in the heap. Our heap is storing vertices. It's never gonna have more than N things in it. So the running time of every heap operation is big O of log N. So what are the heap operations that we do. Well, we extract men and we do it once per iteration of the wild loop. So there's N minus one iterations of the wild loop, just like before, but now instead of doing an exhaustive search through the edges, we just do a simple extract men from the heap and it gives us on a silver platter the vortex we should add next. So what do we do beside extract mins? Well, we have to do this work paying the piper. We have to maintain invariant #two. And every time we extract a min, that then triggers some subsequent key updates. And remember, each of these key updates is a deletion of an element, from the heap followed by an insertion. So how many deletions and insertions do we do? Well, at first this might seem a little bit scary. Right? Because we do a roughly linear number of extract mins. And a vertex might have as many as N-1 outgoing arcs. So it seems like a vertex could trigger as many as N-1 key updates, which is theta of N [inaudible] operations. And if we sum that up over the N iterations of the wild loop that w ould give us N squared heap operations. So, and indeed, in dense graphs, that can be the case. It is true that a single vertex might trigger a linear in N [inaudible] number of [inaudible] operations. But that's the wrong way to think about it. Rather than have this vertex-centric perspective on what, who's responsible for heap operations, let's have an edge-centric view. So for each edge at the graph, let's think about when can this be responsible for some heap operations, in particular a decrease in key in the resulting insertion and deletion. If you have an edge and it points from the vertex V to the vertex W. There's actually only one situation in which this edge is going to be responsible for a, a decrease in key. And that's in the case where the tail of the edge, V. Gets sucked into the set X before the head W of this edge gets sucked into the set X. If that happens, if V gets sucked into X and W is still outside of X, then indeed we're gonna have to decrease the key of W, just like we did in the examples. But that's all that's gonna happen: V can only get sucked into X once and never gonna leave it. So it's only responsible for this single decrease in key of its head W. And that's one insertion and one deletion. And in fact, if the endpoints of this edge get sucked into X in the opposite order, if the tail of, excuse me, if the head of this edge W gets sucked into X first. That doesn't even trigger a, a key decrease for V, and V will never have its de key decreased, because of this particular arc, from V to W. So the upshot is that each edge VW of the graph triggers at most one insert delete combo. So what does this mean, this means that the number of heap operations. Is big O of N, that's for the extract mins. Plus big O of M. That's for the insert the leak combos triggered by edges during the decreased keys. Now just to, I'm gonna write this in a, in a simplified way. This is just O of M, the number of edges. And this is because of our assumption that's there's a path to s from every other vertex. If yo u think about it that means that the graph is at least weakly connected if you picked it up it would stay together in one piece. So that means it at least contains a tree, at least an in an undirected sense, which means it contains at least N minus one edges. So we're in the case of weakly connected graphs where N dominates M. M is always as big as N at least up to a plus one. So what that means is the running time of Dijkstra's algorithm, with this heap implementation, is just a log factor larger. Remember, every heap operation takes time logarithmic. So we do a linear in M number of operations; each takes time logarithmic in N. So the running time is M log N. With, I should say, quite good consistence. So this is a really, really impressively fast algorithm, for computer such a useful problem as shortest paths. So we got a little bit spoiled in our discussion of graph searching connectivity, where it seemed any problem we cared about we could solve in linear time, over M plus N. So here we're picking up this extra logarithmic factor, but I mean, come on, this is still awesome. A running time of M log N is unbelievably faster than a running time of M times N, which is what we had in the straightforward implementation. So this deft use of the heap data structure has given us a truly blazingly fast algorithm for an extremely well motivated problem, computing shortest paths.

# zwk6-Optional

1. In lecture we define the length of a path to be the sum of the lengths of its edges. Define the bottleneck of a path to be the maximum length of one of its edges. A mininum-bottleneck path between two vertices s and t is a path with bottleneck no larger than that of any other s-t path. Show how to modify Dijkstra's algorithm to compute a minimum-bottleneck path between two given vertices. The running time should be O(mlogn), as in lecture. 2. We can do better. Suppose now that the graph is undirected. Give a linear-time (O(m)) algorithm to compute a minimum-bottleneck path between two given vertices. 3. What if the graph is directed? Can you compute a minimum-bottleneck path between two given vertices faster than O(mlogn)?

# zwk6-prog

In this programming problem you'll code up Dijkstra's shortest-path algorithm. Download the following text file: dijkstraData.txt

The file contains an adjacency list representation of an undirected weighted graph with 200 vertices labeled 1 to 200. Each row consists of the node tuples that are adjacent to that particular vertex along with the length of that edge. For example, the 6th row has 6 as the first entry indicating that this row corresponds to the vertex labeled 6. The next entry of this row "141,8200" indicates that there is an edge between vertex 6 and vertex 141 that has length 8200. The rest of the pairs of this row indicate the other vertices adjacent to vertex 6 and the lengths of the corresponding edges.

Your task is to run Dijkstra's shortest-path algorithm on this graph, using 1 (the first vertex) as the source vertex, and to compute the shortest-path distances between 1 and every other vertex of the graph. If there is no path between a vertex v and vertex 1, we'll define the shortest-path distance between 1 and v to be 1000000.

You should report the shortest-path distances to the following ten vertices, in order: 7,37,59,82,99,115,133,165,188,197. You should encode the distances as a comma-separated string of integers. So if you find that all ten of these vertices except 115 are at distance 1000 away from vertex 1 and 115 is 2000 distance away, then your answer should be 1000,1000,1000,1000,1000,2000,1000,1000,1000,1000. Remember the order of reporting DOES MATTER, and the string should be in the same order in which the above ten vertices are given. The string should not contain any spaces. Please type your answer in the space provided.

IMPLEMENTATION NOTES: This graph is small enough that the straightforward O(mn) time implementation of Dijkstra's algorithm should work fine.

OPTIONAL: For those of you seeking an additional challenge, try implementing the heap-based version. Note this requires a heap that supports deletions, and you'll probably need to maintain some kind of mapping between vertices and their positions in the heap.