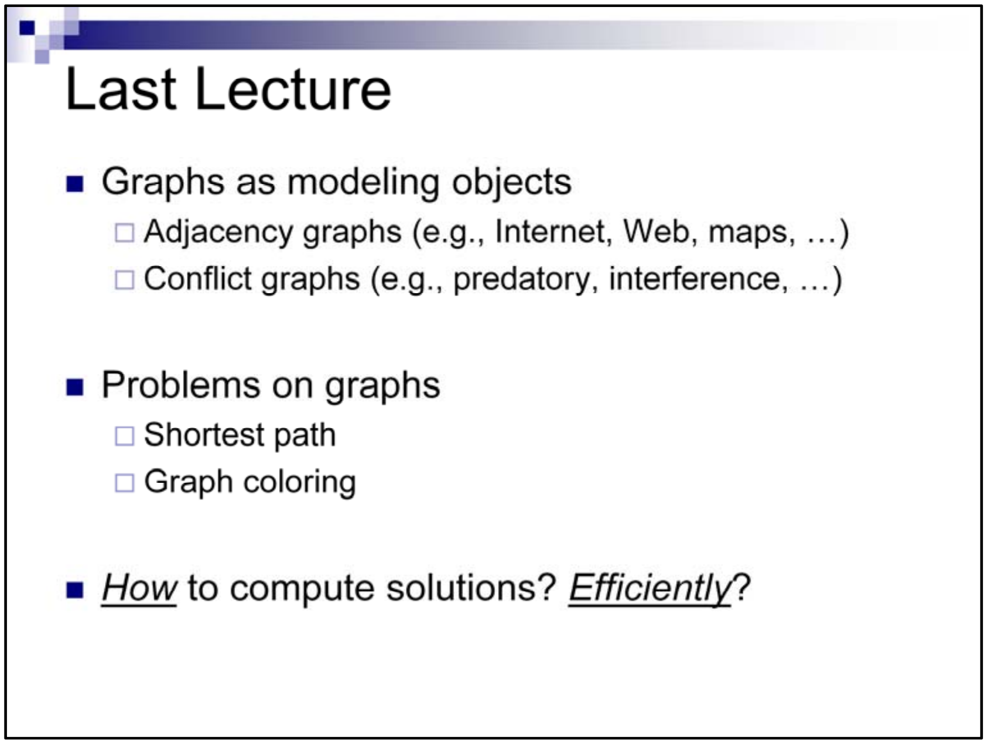
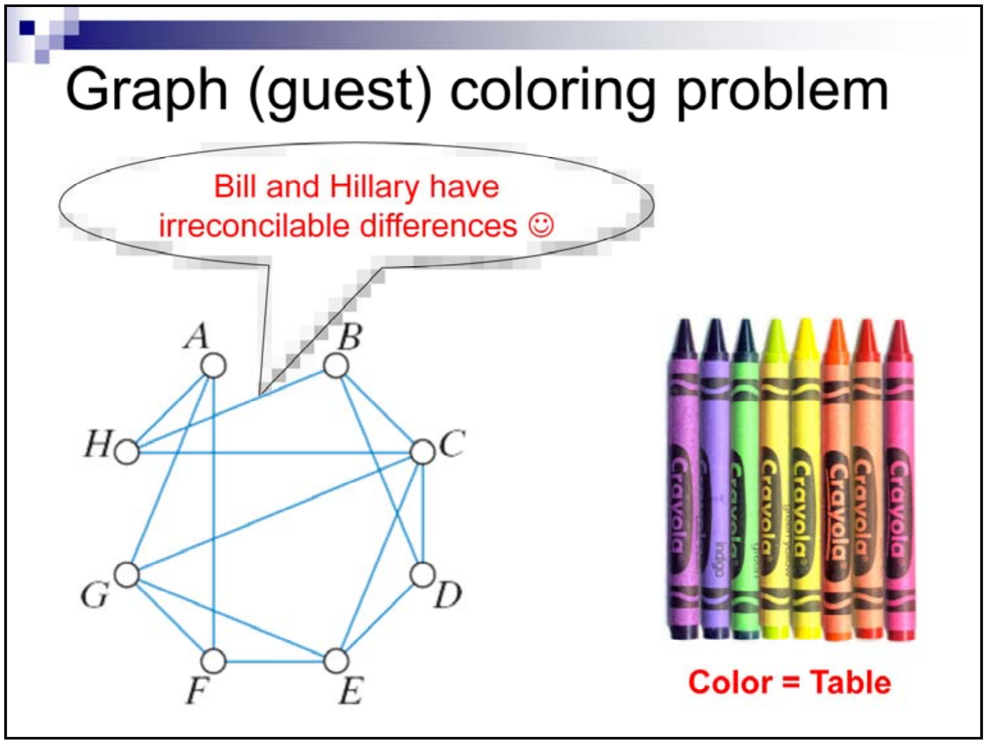
MA/CS-109: Shortest Path Computation

Azer Bestavros



Last Lecture

- Graphs as modeling objects
  - Adjacency graphs (e.g., Internet, Web, maps, …)
  - Conflict graphs (e.g., predatory, interference, …)
- Problems on graphs
  - Shortest path
  - Graph coloring
- *How* to compute solutions? *Efficiently*?



Graph (guest) coloring problem

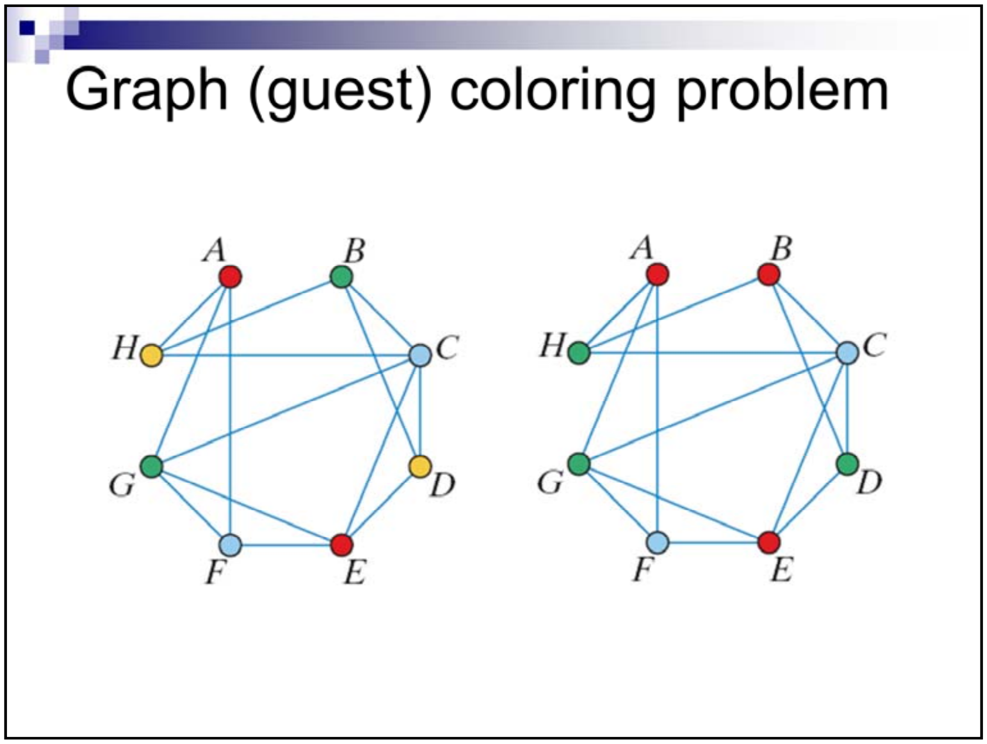
Bill and Hillary have irreconcilable differences ☺

Color = Table

Yet, another example is the problem of arranging guests attending a wedding around tables. Given a list of “irreconcilable differences” between invitees, you task is to find a seating arrangement that uses the least number of tables while avoiding to have two individuals with “irreconcilable differences” sitting around the same table. This is a graph coloring problem! Namely, we can solve the problem by (1) modeling each individual as a vertex in a graph, (2) representing the fact that two individuals have irreconcilable differences by drawing an edge between the vertices corresponding to these individuals, and (3) minimizing the number of colors (tables) used to color the vertices (seat the guests) such that no two vertices connected with an edge (two guests with irreconcilable differences) have the same color (are sitting on the same table).

Along the same lines, one can use graph vertex coloring to solve other (quite important) problems, including assigning the minimum number of radio frequencies to various radio transmitters who may interfere with one another – here the radio stations would be the vertices, and if two stations interfere with one another (because of geographical proximity) then we draw an edge between them. Now coloring the graph is akin to assigning frequencies since we would not want to give the same frequency (color) to interfering stations (adjacent vertices).

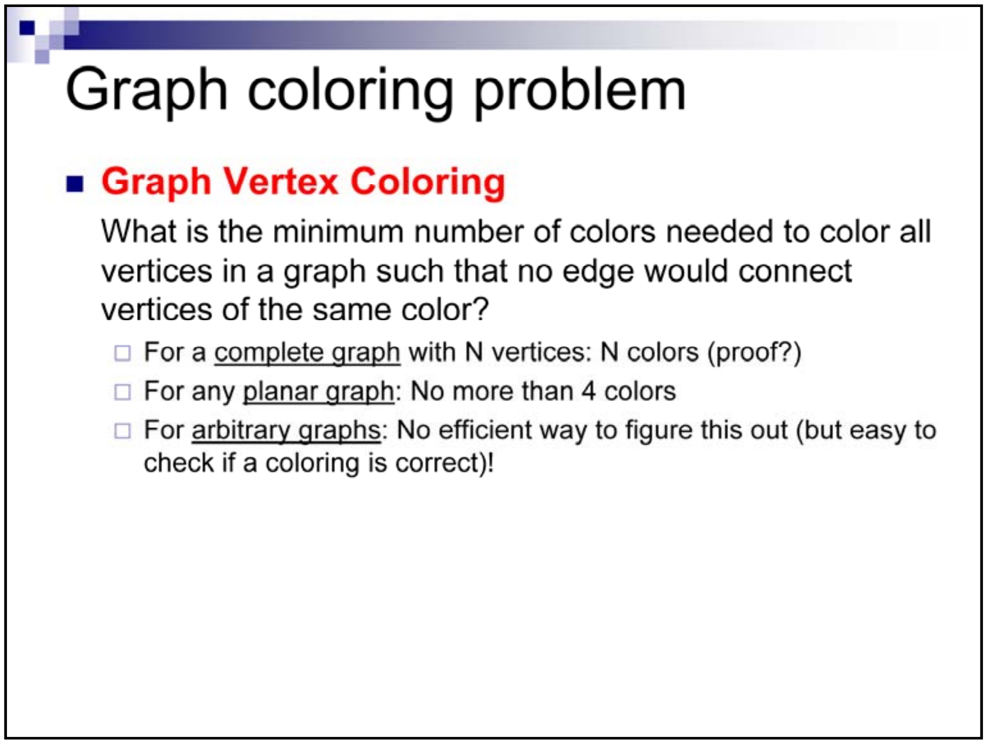
Yet, another famous application of graph coloring is “map coloring”. Here our job is to color adjacent states (vertices) on a planar map (e.g., map of the US) with different colors , but use the minimum number of colors. This fairly classical problem was settled by proving that one need no more than 4 colors for such (planar) graphs.



Graph (guest) coloring problem

The attempt on the left shows a coloring (assignment of guests to tables) that uses 4 colors (i.e., we need 4 tables). The one on the right shows a coloring that uses only 3 colors. One can show that for this graph, three is the minimum number of colors. Try to prove it!

Accordingly, we would seat A, B, and E on the first (red) table; D, G, and H on the second (green) table; and C, and F on the third (blue) table.



Graph coloring problem

- **Graph Vertex Coloring**

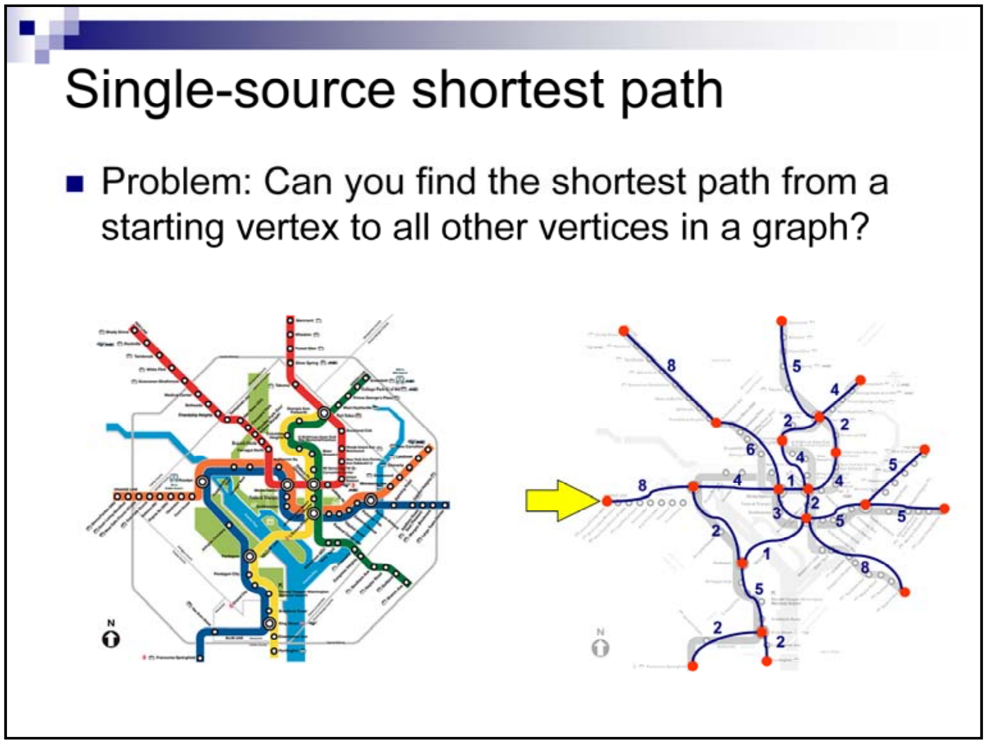
  What is the minimum number of colors needed to color all vertices in a graph such that no edge would connect vertices of the same color?
  - For a complete graph with N vertices: N colors (proof?)
  - For any planar graph: No more than 4 colors
  - For arbitrary graphs: No efficient way to figure this out (but easy to check if a coloring is correct)!

So, how many colors do we need?

Well, the answer is relatively easy for some special graphs:

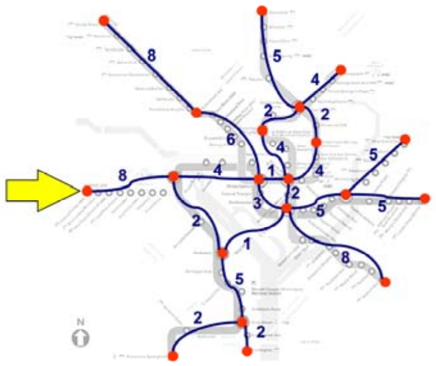
1. For a complete graph with N nodes (a graph where each pair of vertices are connected with an edge), we would obviously need N colors. One can prove this very easily by contradiction – Assume that one can color a complete graph with less than N colors such that no two adjacent vertices have the same color. Since the graph has N vertices, then it must be the case that at least two vertices have the same color. But since there is an edge between any pair of vertices, it follows that two adjacent vertices have the same color, which is a contradiction.
2. For a ring with N nodes (a graph where each vertex is connected to exactly one other vertex forming a connected cycle). One can prove that if N is even, then one needs two colors, and if N is odd, one needs 3 colors.
3. For a planar graph (a graph that can be drawn on a plane without having any of the graph's edges intersect), as we mentioned before, it was shown that one needs at most 4 colors…

How about arbitrary graphs? Unfortunately, one can show that finding out the minimum number of colors for arbitrary graphs cannot be done in an efficient way (unless the graph is relatively small).

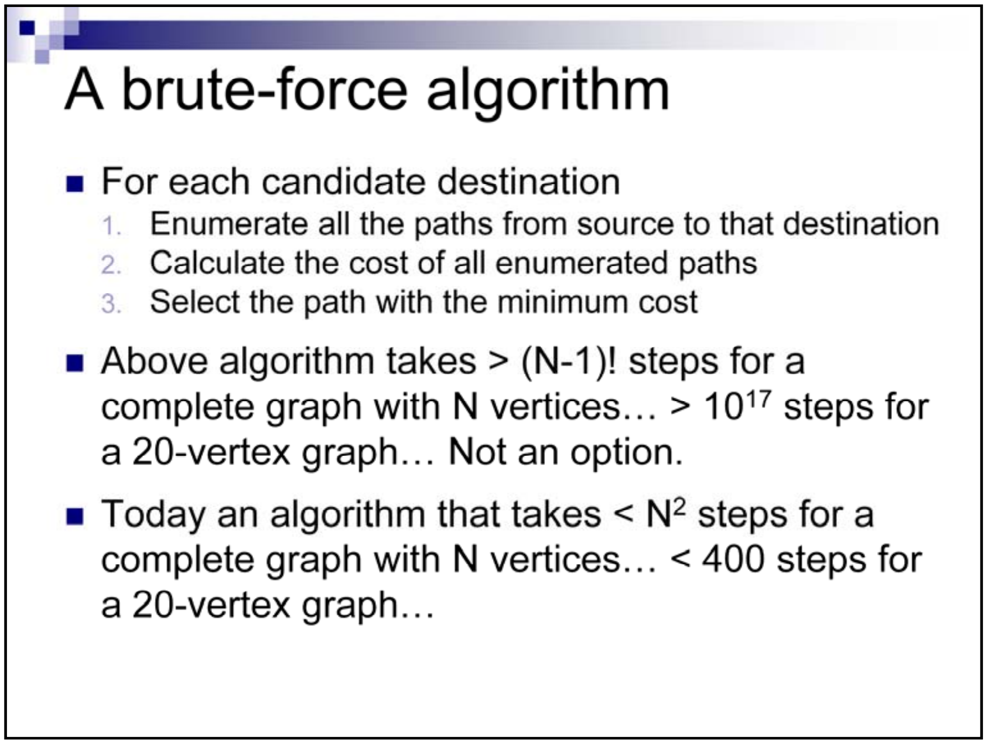


Single-source shortest path

- Problem: Can you find the shortest path from a starting vertex to all other vertices in a graph?



Recall the problem of finding the shortest path from a starting vertex to all other vertices in a graph (modeling a real-world artifact)…

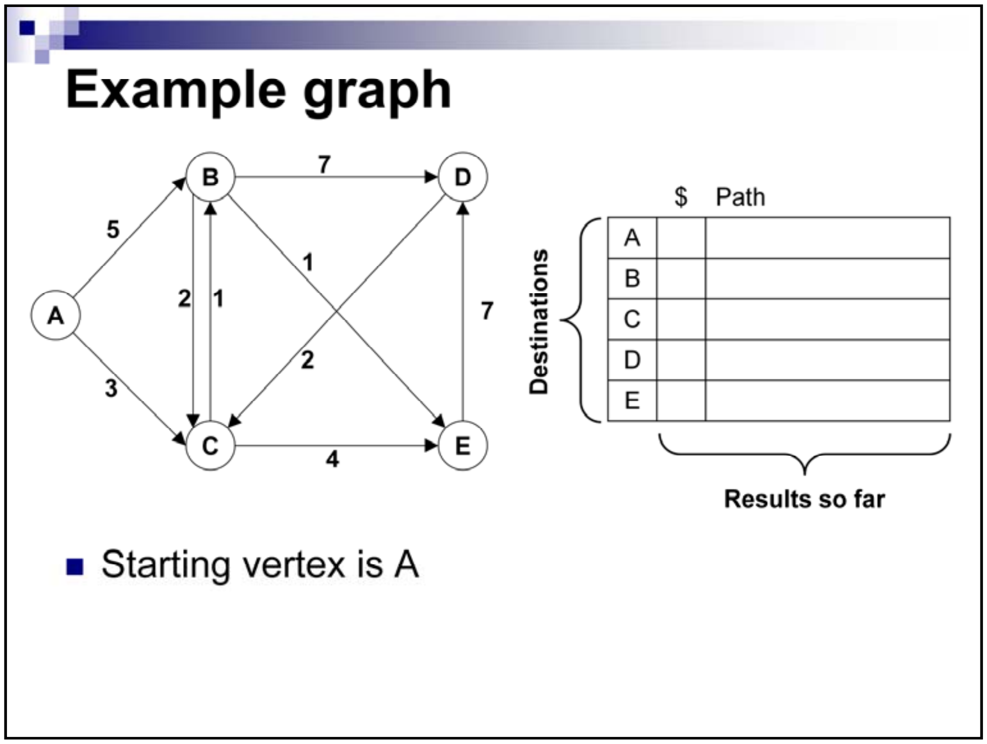


A brute-force algorithm

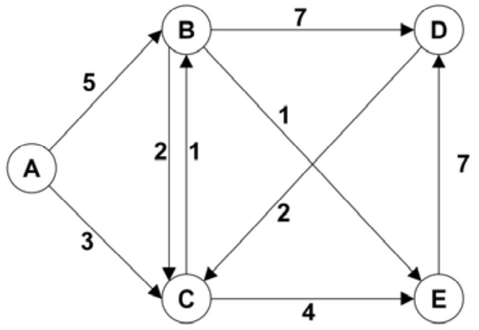
- For each candidate destination
  1. Enumerate all the paths from source to that destination
  2. Calculate the cost of all enumerated paths
  3. Select the path with the minimum cost
- Above algorithm takes > (N-1)! steps for a complete graph with N vertices… > 10^{17} steps for a 20-vertex graph… Not an option.
- Today an algorithm that takes < N^2 steps for a complete graph with N vertices… < 400 steps for a 20-vertex graph…

Last time we discussed a "brute-force" approach that consists of (1) enumerating all the paths from the starting point, (2) figuring out how much each such path costs, and then (3) for each destination selecting the path with the minimum cost. This approach is not really practical, in terms of how long it would take to do all this for graphs of sizes as small as (say) 20.

Our goal in this lecture is to come up with an "efficient" algorithm…



Example graph



Destinations

	$	Path
A		
B		
C		
D		
E		

Results so far

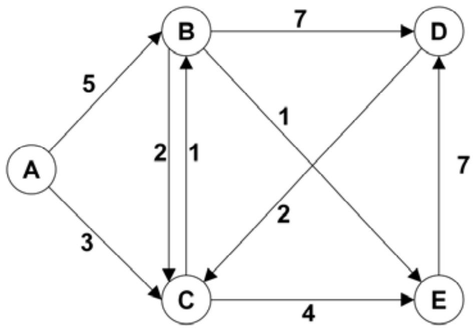
- Starting vertex is A

The best way to do this is to work on a simple example (such as the one shown above) and see if we can come up with a procedure.

To be able to carry out our procedure, we would need some scratch space to keep track of paths we discover, their costs, etc.

In particular, our scratch space (in computer science lingo, this is called a data structure) is a table with one row per vertex (destination). For our example, we would need a table with 5 rows for vertices A, B, C, D, and E. In the table we will keep track of two pieces of information: The best path we have discovered (so far) and the corresponding cost.

Initialization



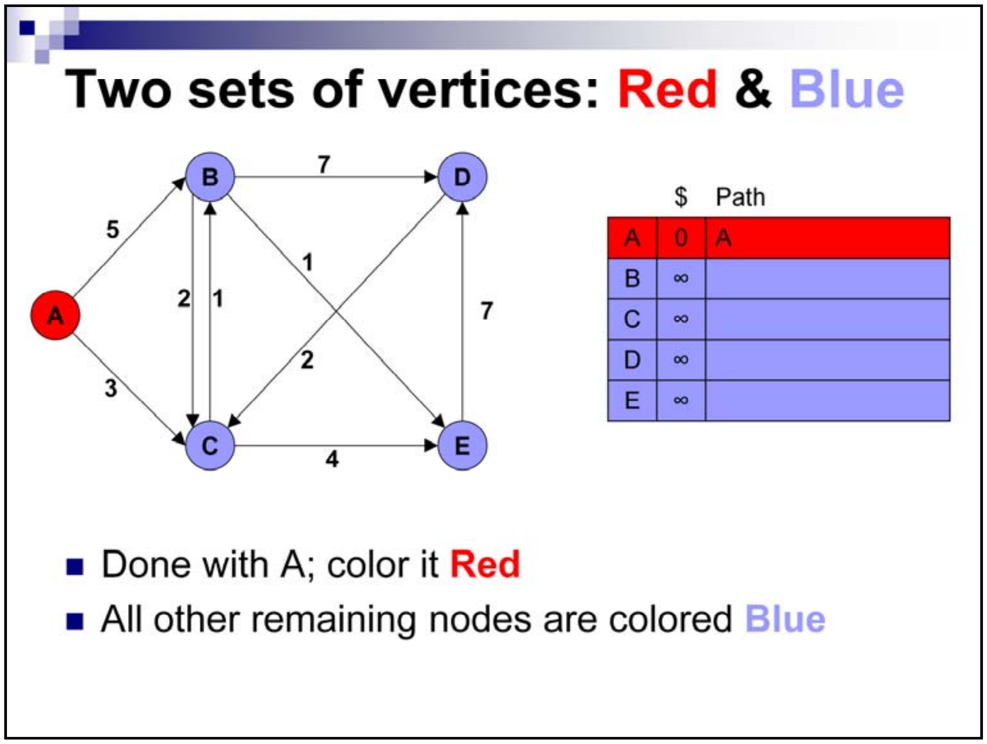
	$	Path
A	0	A
B	∞	
C	∞	
D	∞	
E	∞	

- Start vertex is A; Cost to A = 0; Path to A is trivial as we are there already.
- Paths to all other destinations are not known; Cost of getting to those destination is infinite

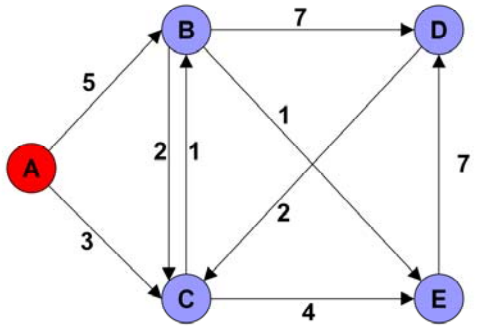
Next, we would need to initialize our table.

In particular, it is obvious that if we pick A (the start vertex) as our destination, then we just stay where we are, and the cost of doing so is zero! Thus, we initialize the cost and path for A accordingly.

For all other vertices, since at this stage we do not know how to get to any of them (from A), we initialize their costs to infinity.



Two sets of vertices: Red & Blue



	$	Path
A	0	A
B	∞	
C	∞	
D	∞	
E	∞	

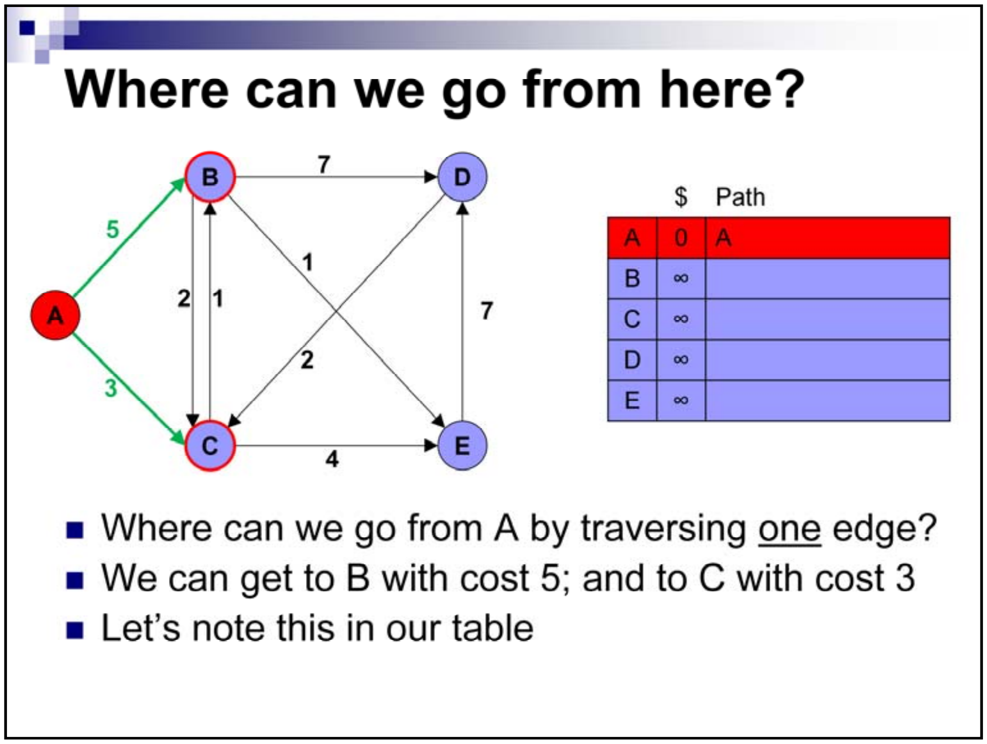
- Done with A; color it **Red**
- All other remaining nodes are colored **Blue**

In our table, we will distinguish between two sets of vertices:

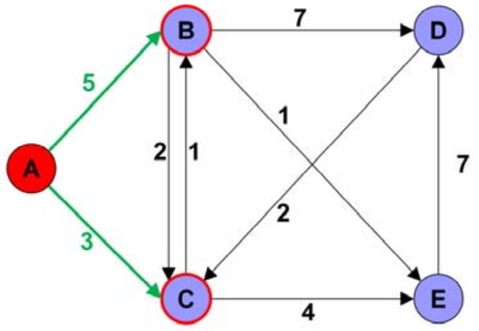
1. Those for which we have computed a (proven) shortest path. We will color these RED. Initially, the start vertex is the only RED vertex
2. Those for which we do not have a (proven) shortest path. We will color these BLUE. Initially, all vertices except the start vertex are BLUE.

Our goal (eventually, by the end of our procedure) is to compute the shortest path for all vertices. In other words, our goal is to turn all BLUE vertices to RED vertices.

We will try to do this incrementally by turning one vertex from BLUE to RED at a time.



Where can we go from here?



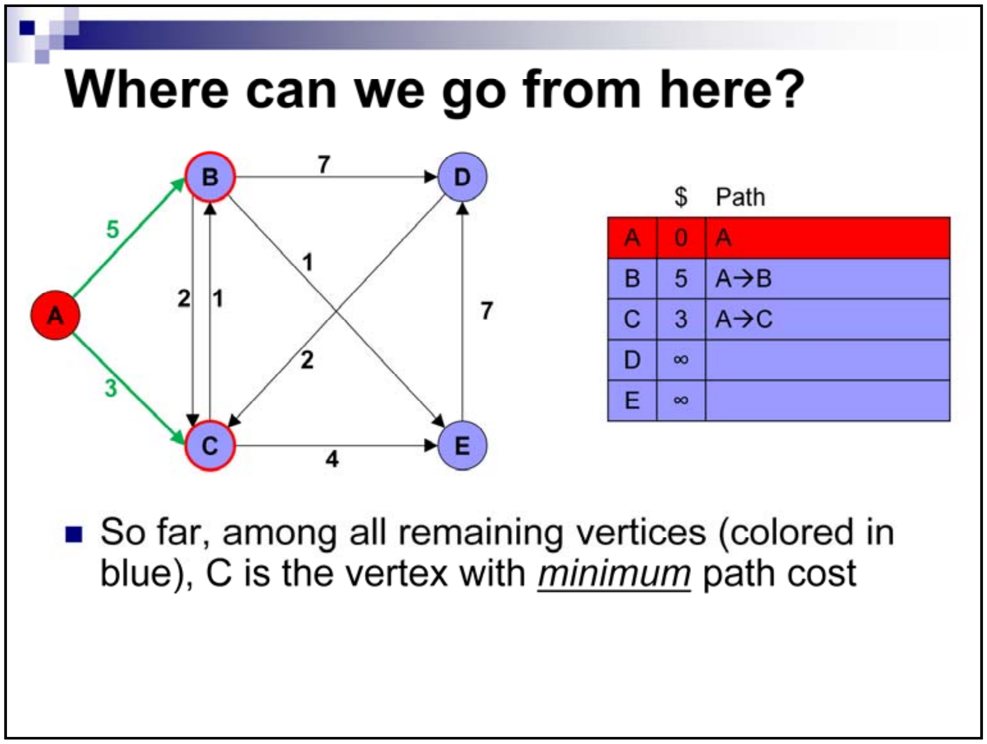
	$	Path
A	0	A
B	∞	
C	∞	
D	∞	
E	∞	

- Where can we go from A by traversing <u>one</u> edge?
- We can get to B with cost 5; and to C with cost 3
- Let's note this in our table

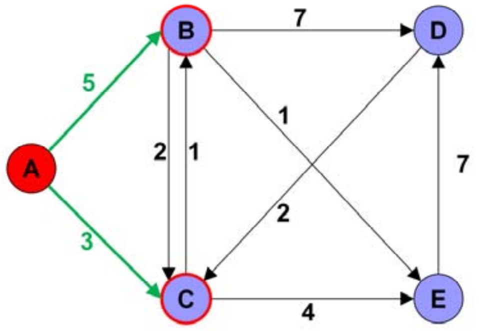
The only way to reach any of the blue vertices from the start vertex is to follow one of the green-colored edges (basically, there is no other way to get to these blue vertices).

So, now, we ask the question, if we traverse exactly <u>one</u> of these edges where would we end up and at what cost?

In our example, we can get to B with cost 5 and to C with cost 3. This gives us two candidate paths that we can now record in our table.



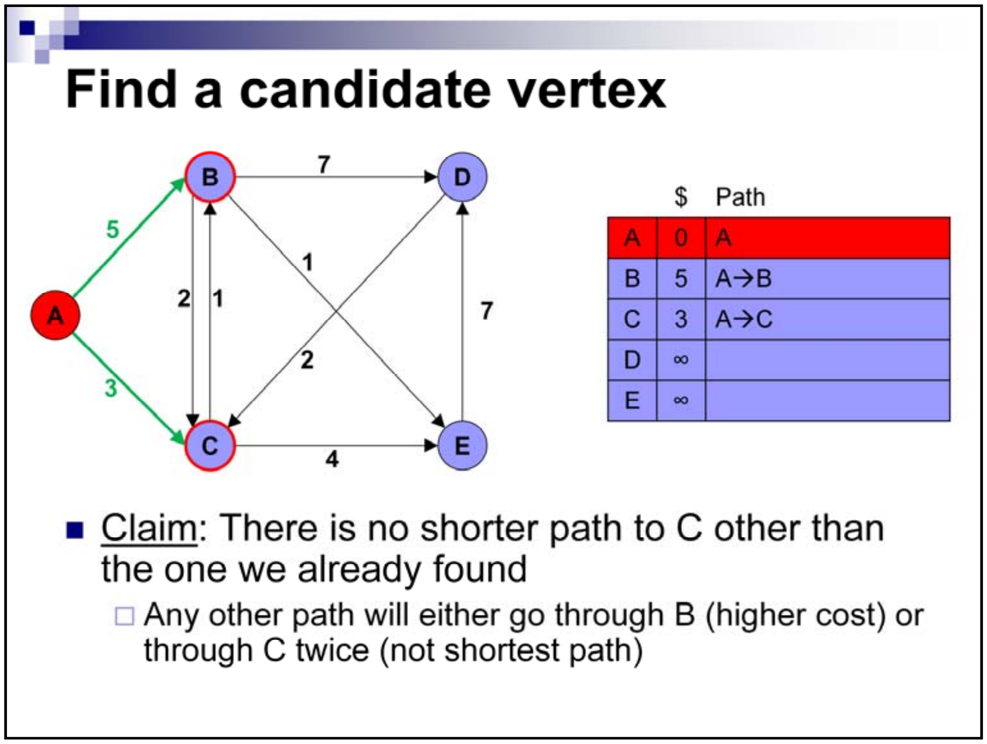
Where can we go from here?



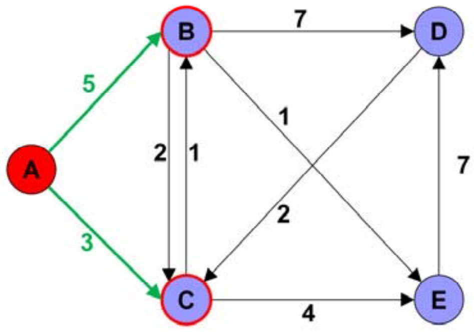
	$	Path
A	0	A
B	5	A→B
C	3	A→C
D	∞	
E	∞	

- So far, among all remaining vertices (colored in blue), C is the vertex with *minimum* path cost

Among all remaining blue vertices, C is a destination we can reach with a minimum cost (according to the costs in our table).



Find a candidate vertex



	$	Path
A	0	A
B	5	A→B
C	3	A→C
D	∞	
E	∞	

- Claim: There is no shorter path to C other than the one we already found
  - Any other path will either go through B (higher cost) or through C twice (not shortest path)

Well, could it be that the path we have for C is the shortest path? Can we prove that it is? The answer is yes!

So, let’s try to prove the conjecture (claim) that *there is no path to C shorter than the one we already found.*

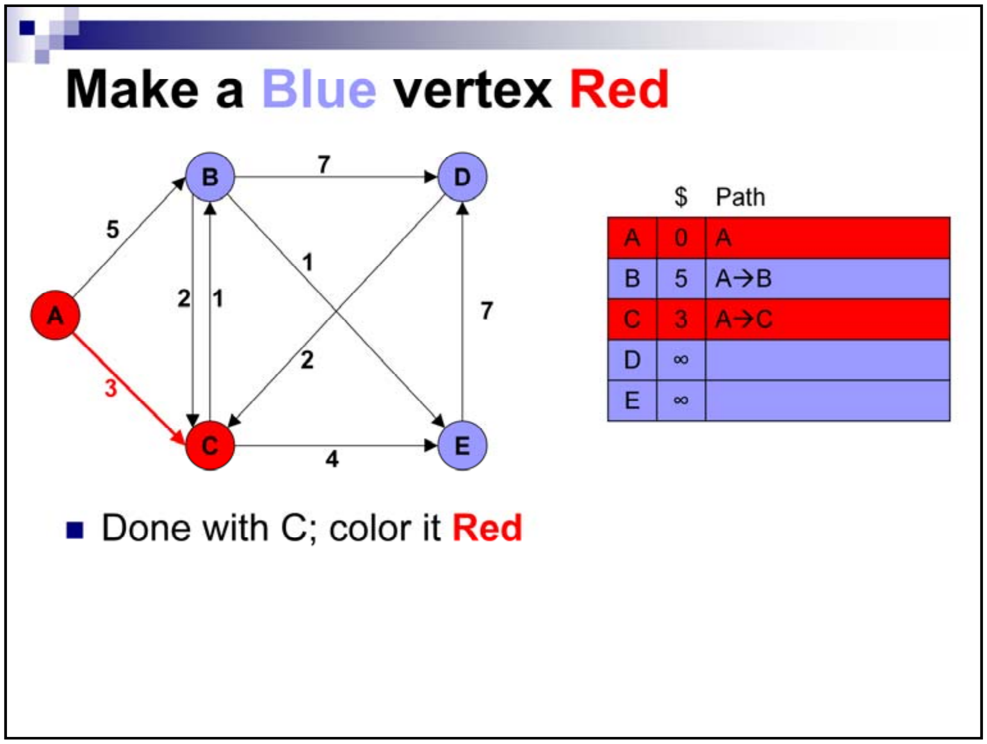
We will prove our conjecture by contradiction.

Assume that there is path shorter than what we already discovered that goes from A to C. What would such a path look like? It would be either

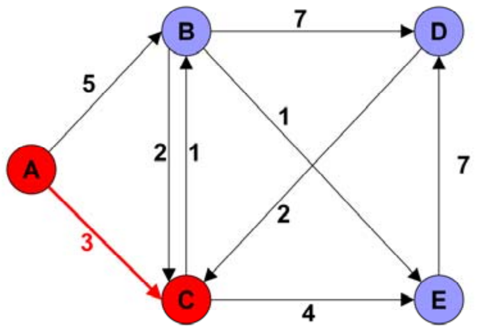
1. A→B→…→C, or
2. A→C→…→C

But both of the above paths must have a cost higher than the one we have (which is A→C). Why? Because going through B will cost more (at least 5 as opposed to our current cost of 3) and going through C twice is bound to be more costly as well since the cost of the cycle (from C back to itself) must have a positive cost since edge costs are positive (we established this last lecture).

By showing that any path other than the one we discovered will be of higher cost, we proved our conjecture!



Make a Blue vertex Red

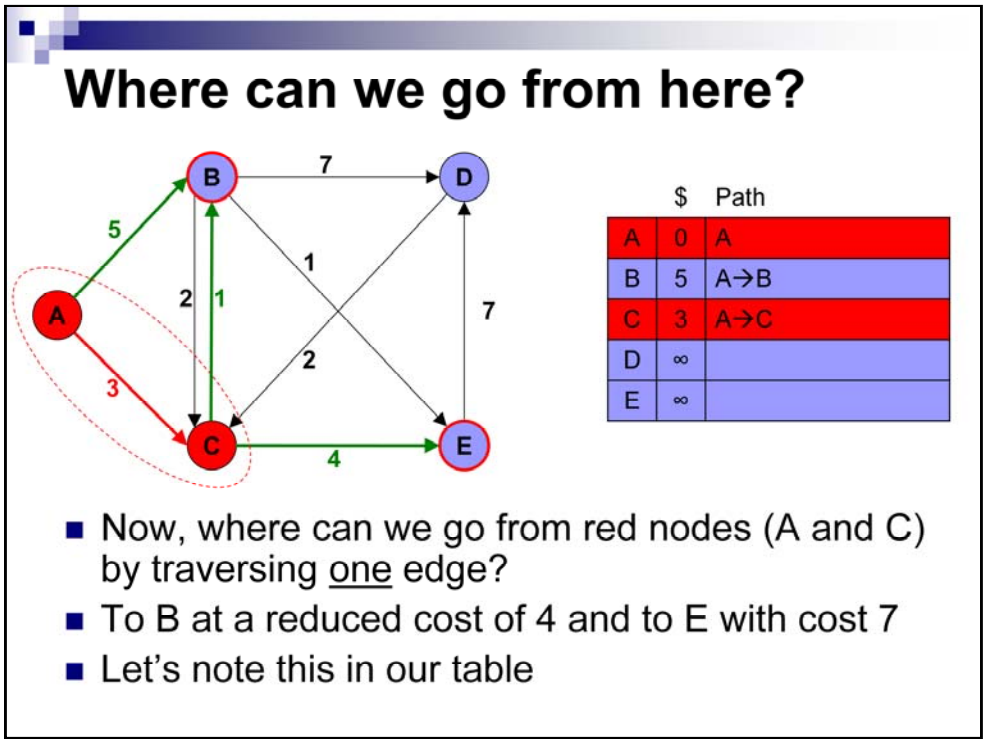


	$	Path
A	0	A
B	5	A→B
C	3	A→C
D	∞	
E	∞	

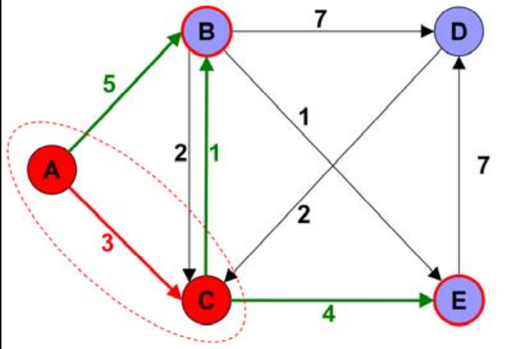
- Done with C; color it **Red**

So, now we know that we have a PROVEN shortest path to C.

Thus, C turns from being a blue vertex to a red vertex.



Where can we go from here?



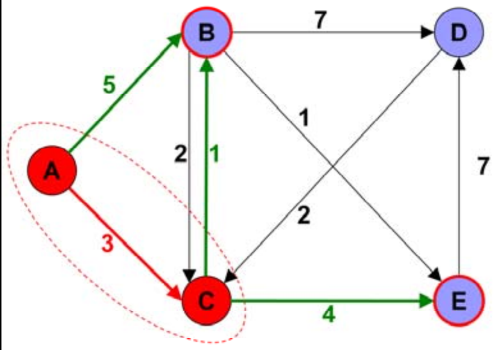
	$	Path
A	0	A
B	5	A→B
C	3	A→C
D	∞	
E	∞	

- Now, where can we go from red nodes (A and C) by traversing one edge?
- To B at a reduced cost of 4 and to E with cost 7
- Let's note this in our table

And, now we repeat the same type of reasoning, asking ourselves what new paths can we evaluate going from red vertices to blue vertices by traversing exactly one edge…

We update our table…

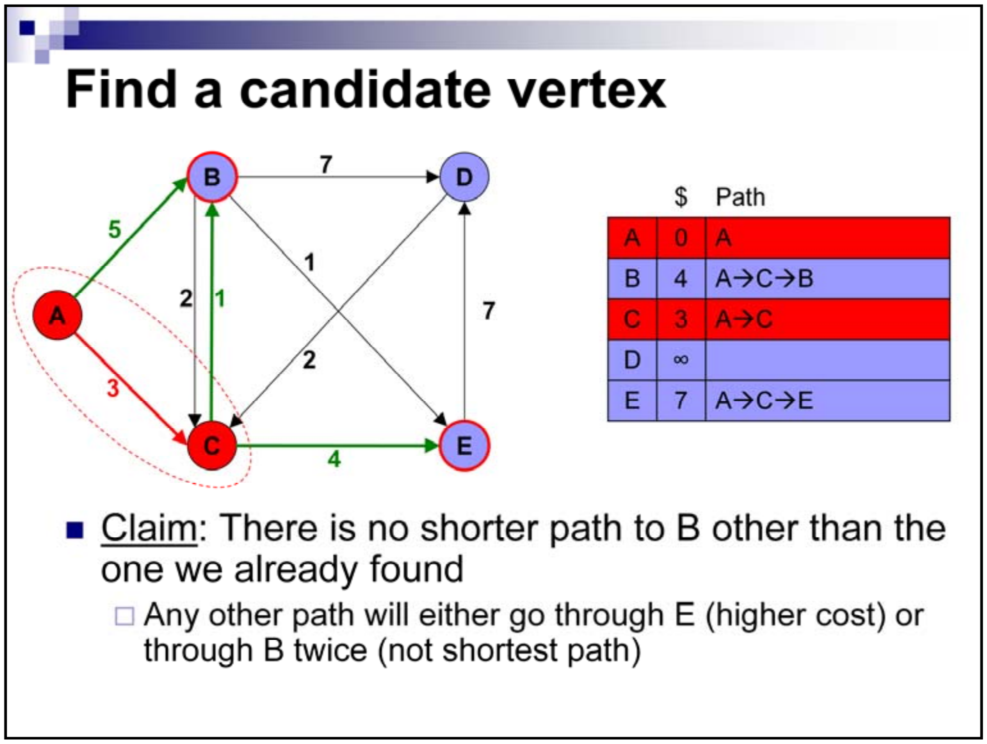
Where can we go from here?



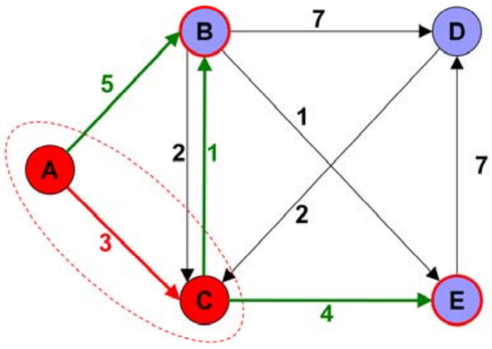
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	∞	
E	7	A→C→E

- So far, among all remaining vertices (colored in blue), B is the vertex with *minimum* path cost

And, as before we identify the blue vertex that we can reach with the minimum cost (according to our current table) – namely B in our example.



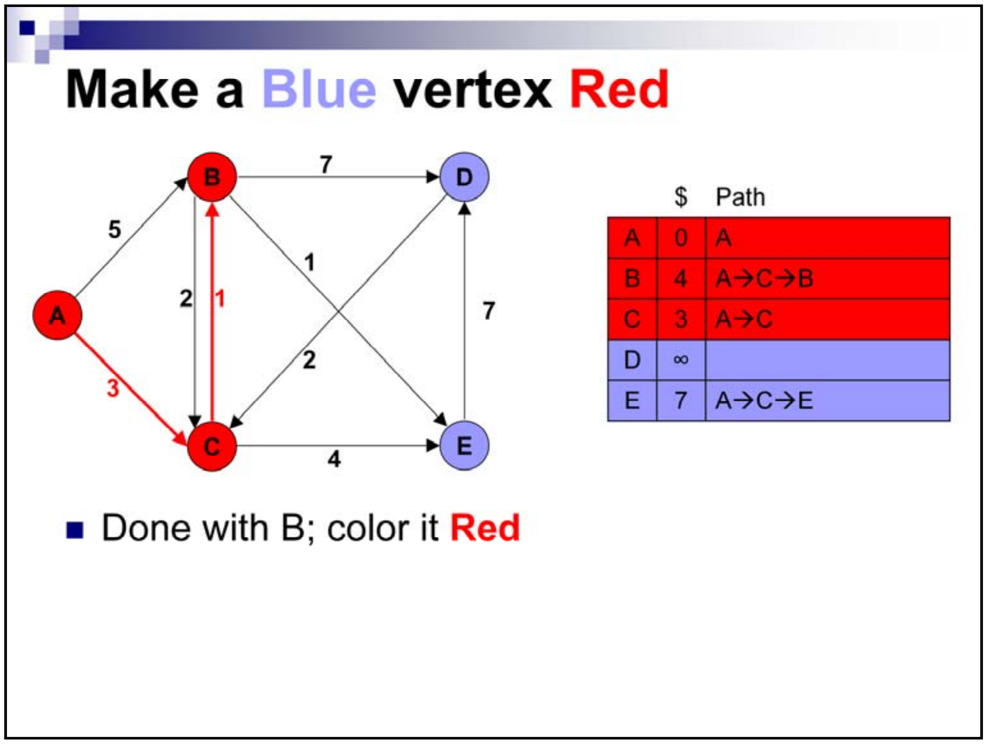
Find a candidate vertex



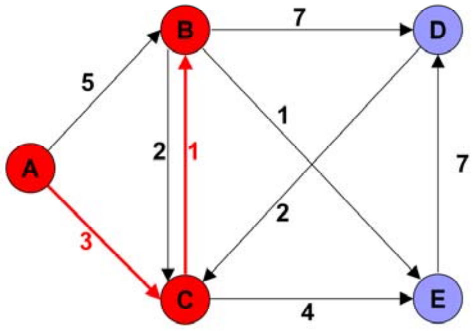
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	∞	
E	7	A→C→E

- Claim: There is no shorter path to B other than the one we already found
  - Any other path will either go through E (higher cost) or through B twice (not shortest path)

Now we make the claim that the path we have to that vertex (B) is the shortest path from A to B -- a claim we can prove by noting that any other path to that vertex will necessarily be more costly.



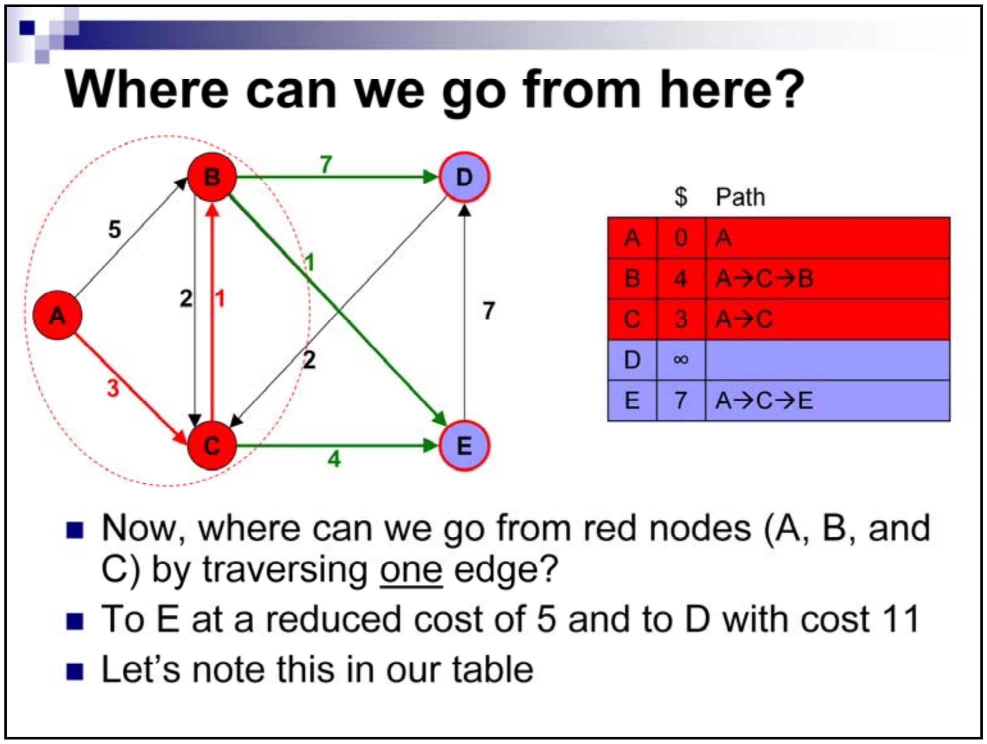
Make a Blue vertex Red



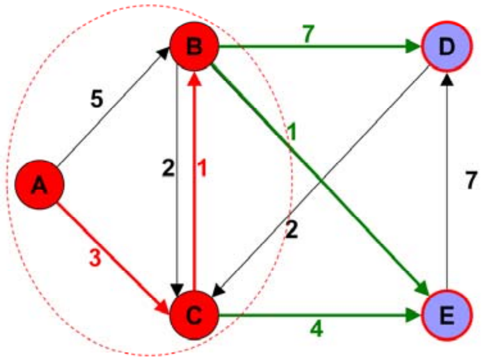
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	∞	
E	7	A→C→E

- Done with B; color it **Red**

Which leads us to turn one more node (B) from blue to red.



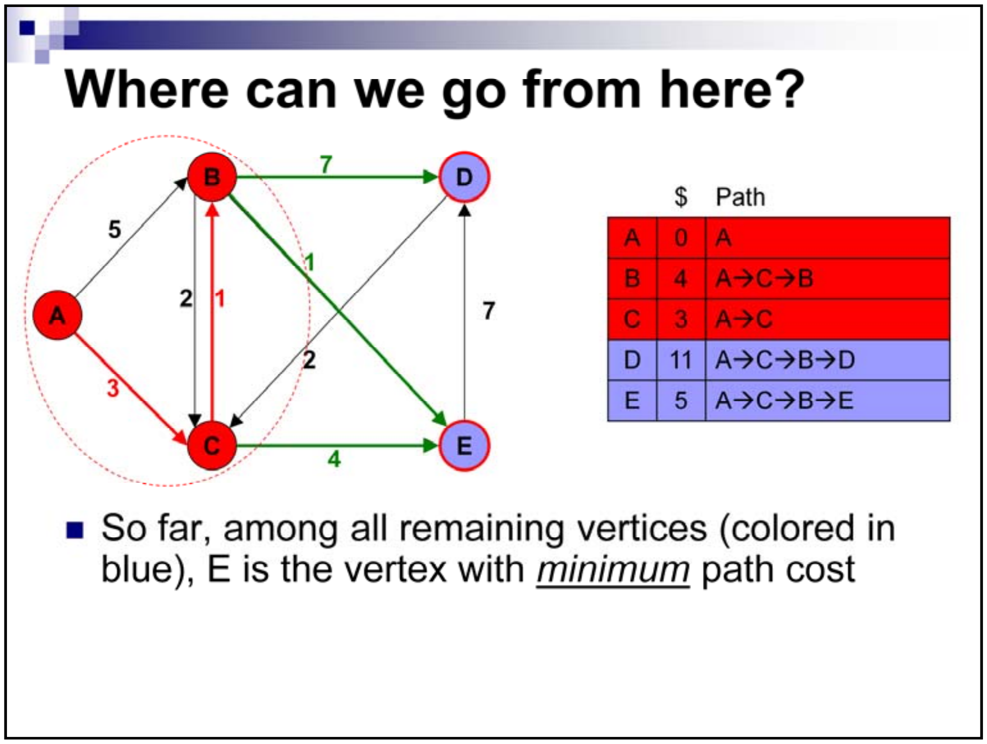
Where can we go from here?



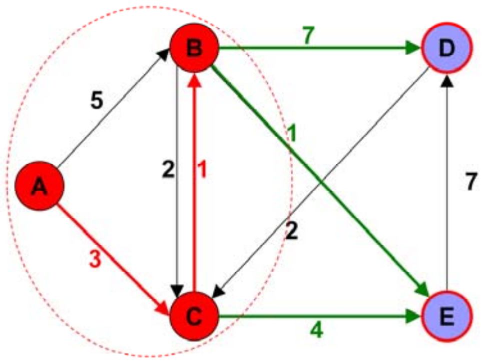
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	∞	
E	7	A→C→E

- Now, where can we go from red nodes (A, B, and C) by traversing one edge?
- To E at a reduced cost of 5 and to D with cost 11
- Let's note this in our table

And the process continues, by identifying new paths that we evaluate by traversing exactly one of the edges from the red vertices to the blue vertices, updating the table accordingly…



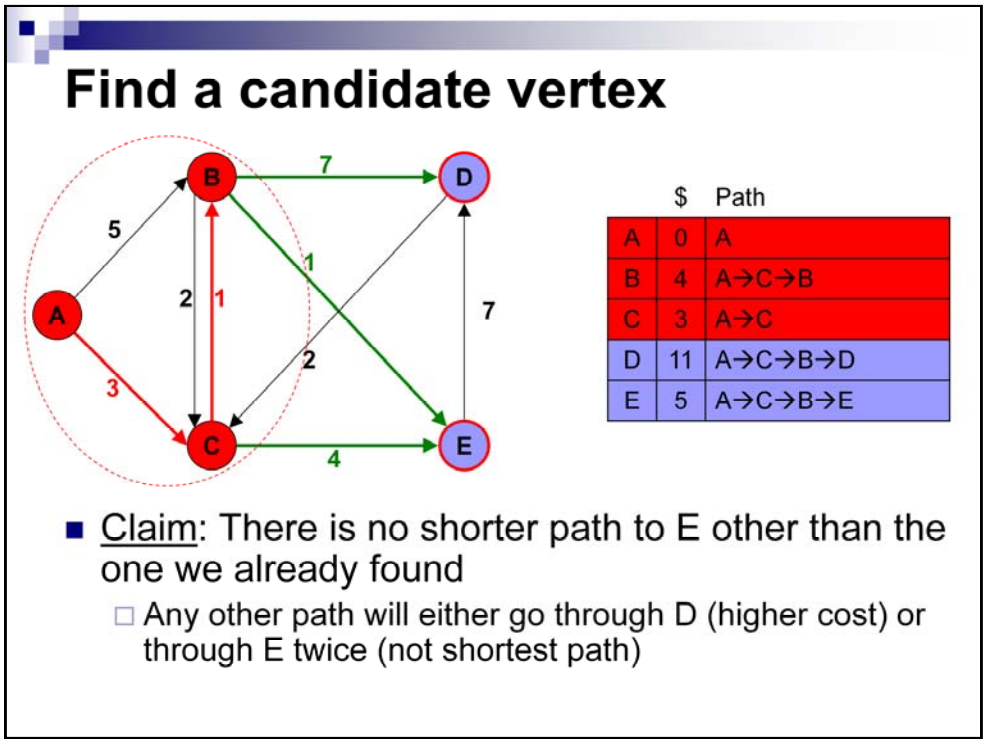
Where can we go from here?



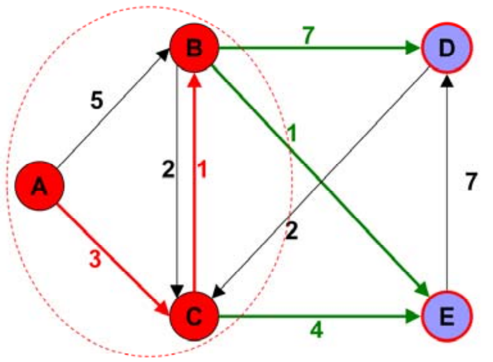
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- So far, among all remaining vertices (colored in blue), E is the vertex with *minimum* path cost

… identifying a candidate vertex to turn red (a blue vertex with the minimum cost in the table) -- namely E.



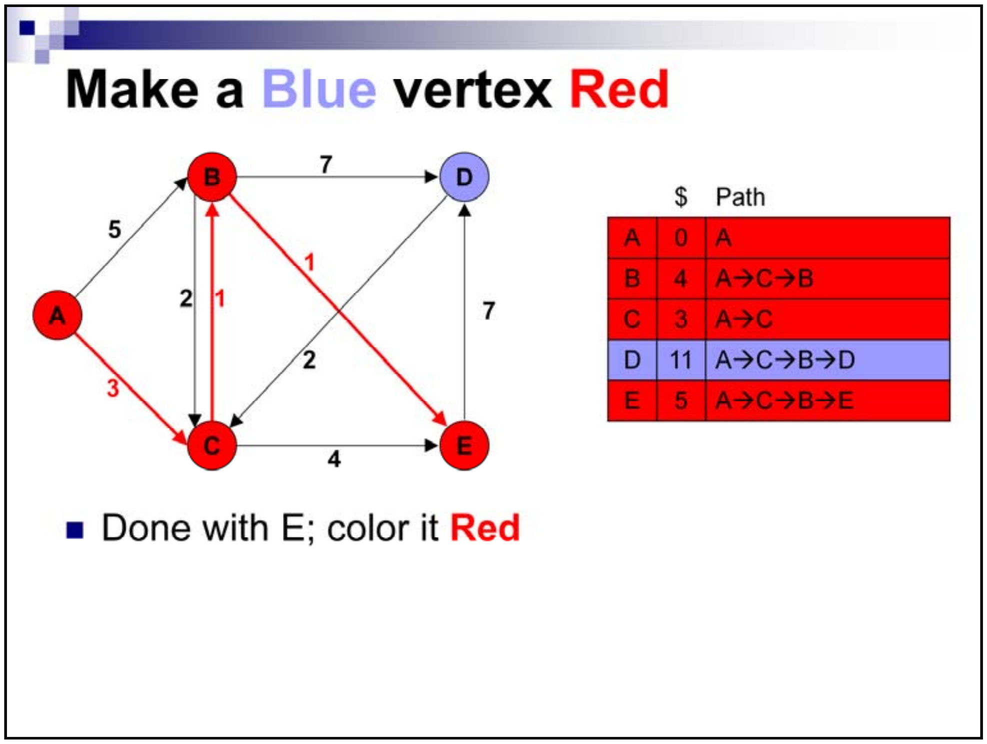
Find a candidate vertex



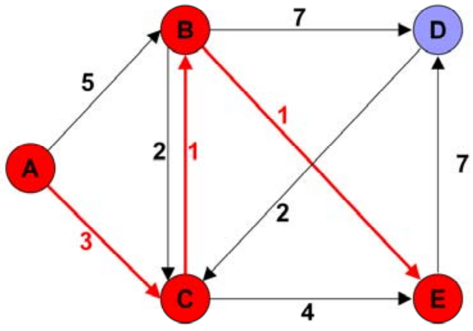
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- Claim: There is no shorter path to E other than the one we already found
  - Any other path will either go through D (higher cost) or through E twice (not shortest path)

The path we have to E is the shortest path, so …



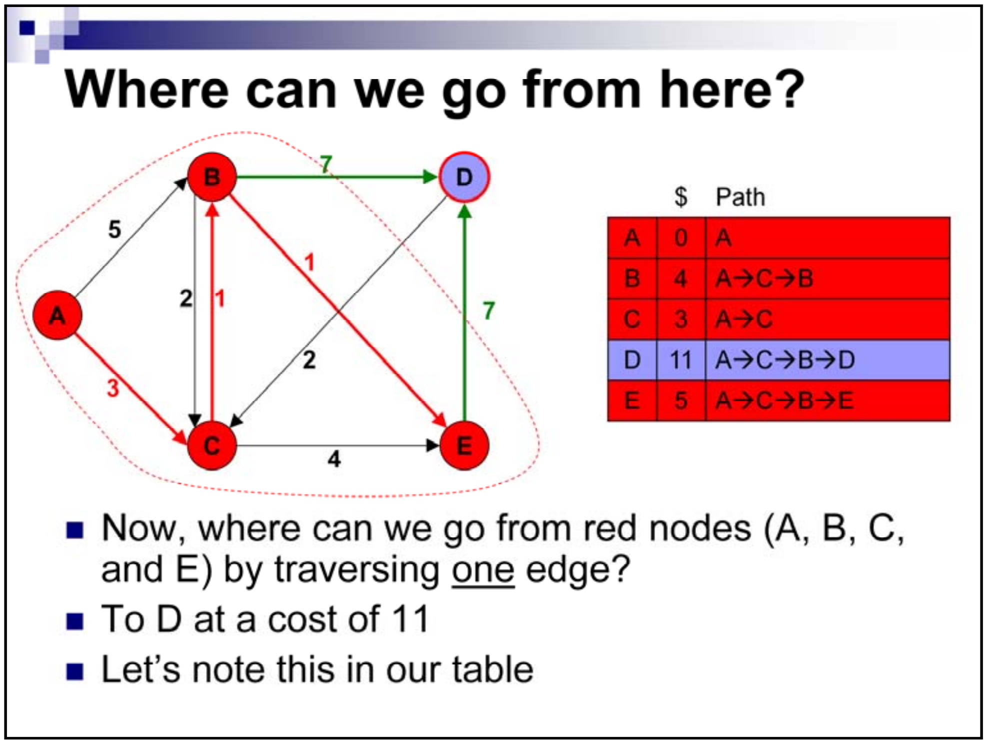
Make a Blue vertex Red



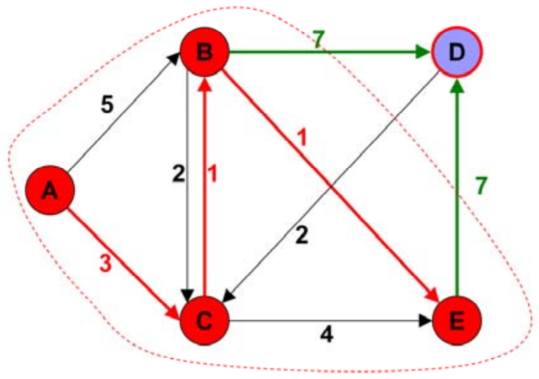
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- Done with E; color it **Red**

… we color E red.



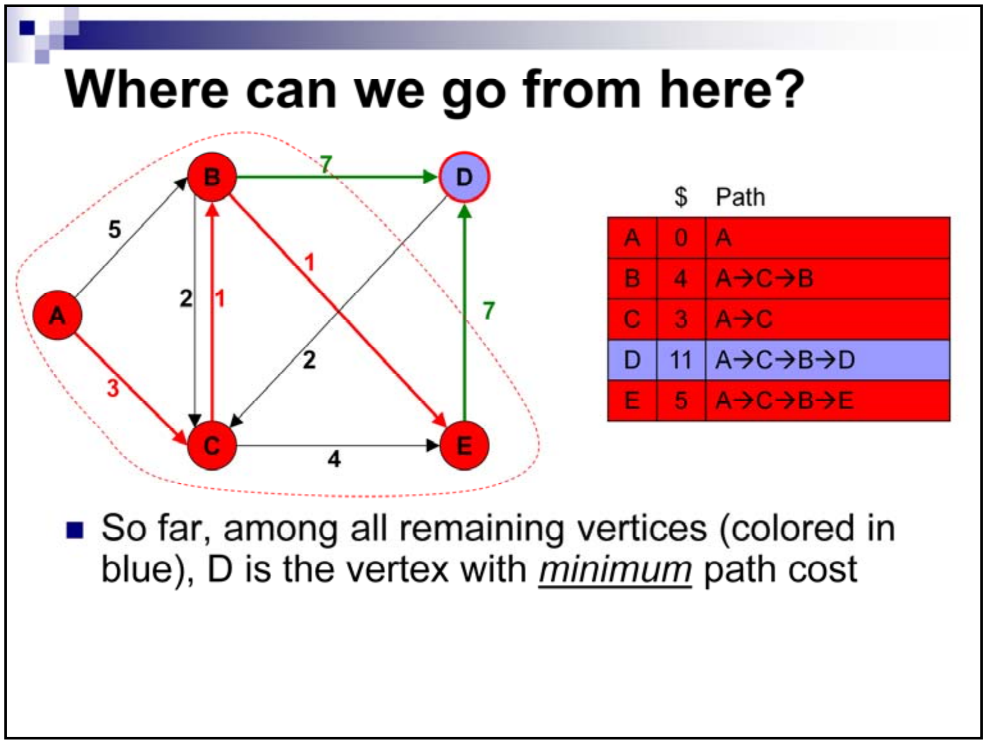
Where can we go from here?



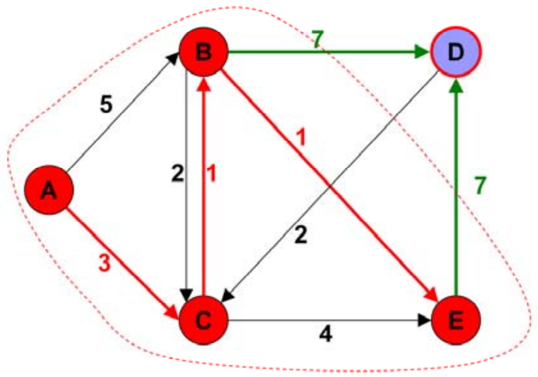
	$	Path
A	0	A→
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- Now, where can we go from red nodes (A, B, C, and E) by traversing one edge?
- To D at a cost of 11
- Let's note this in our table

Now we know the drill…



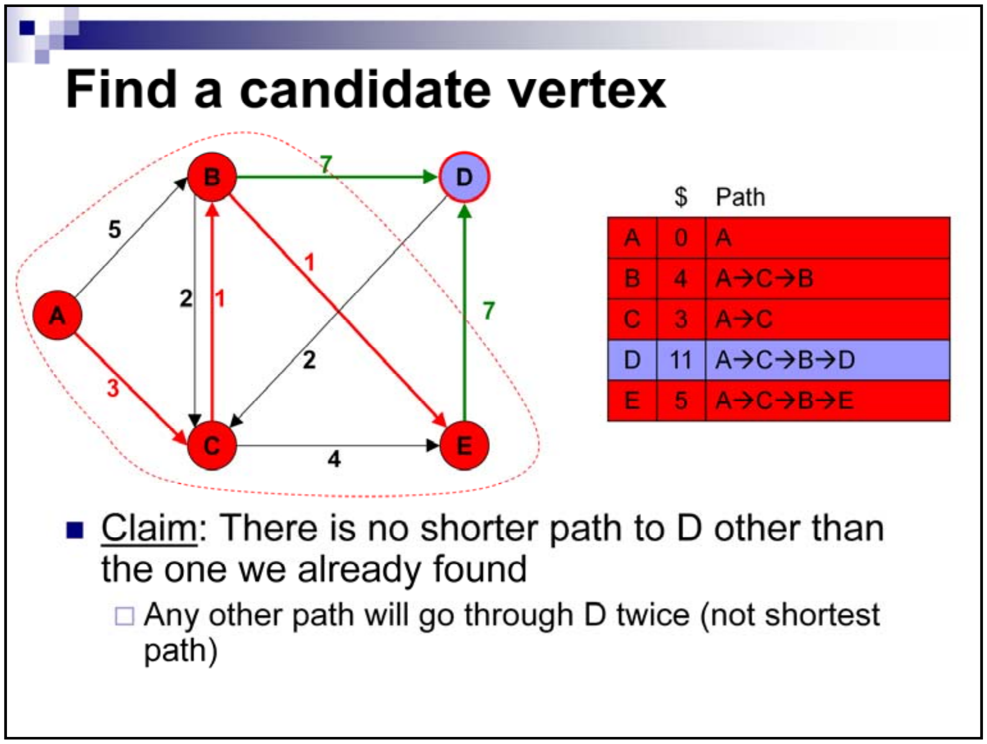
Where can we go from here?



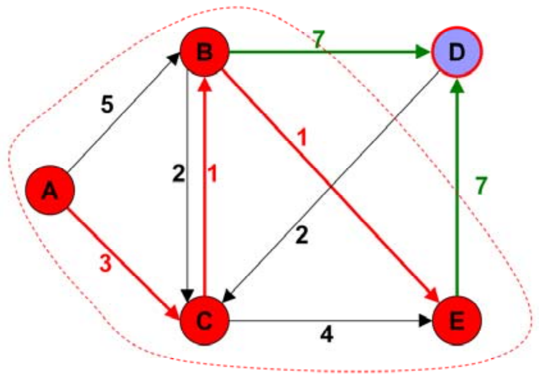
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- So far, among all remaining vertices (colored in blue), D is the vertex with *minimum* path cost

Now we know the drill…



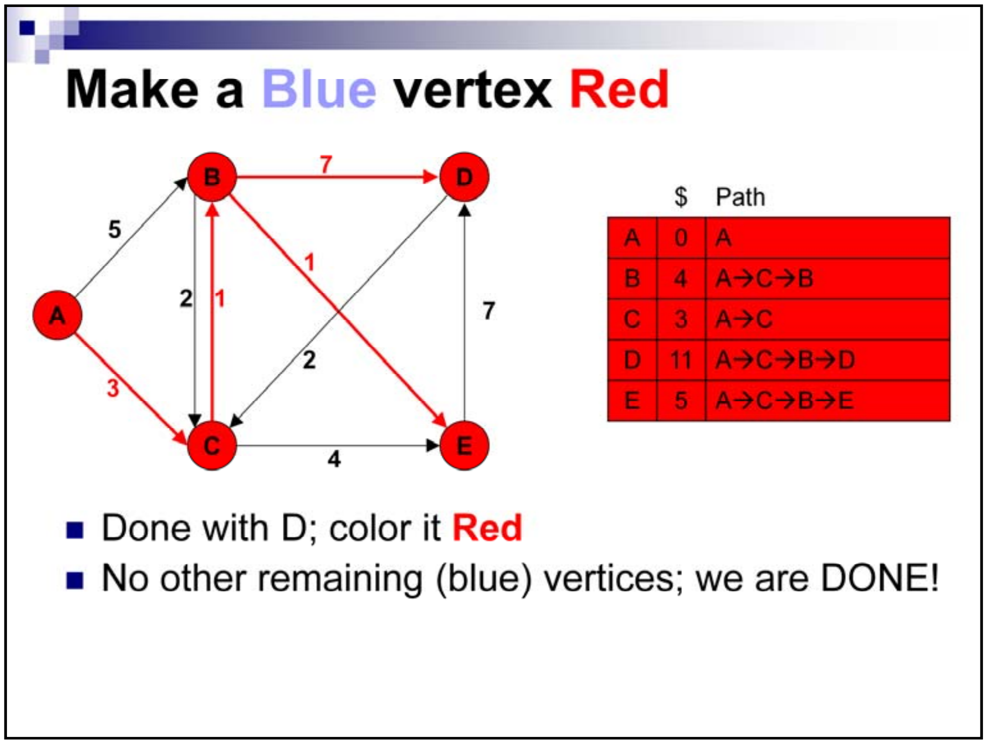
Find a candidate vertex



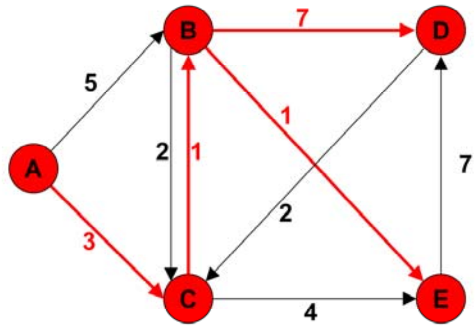
	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- Claim: There is no shorter path to D other than the one we already found
  - Any other path will go through D twice (not shortest path)

Now we know the drill…



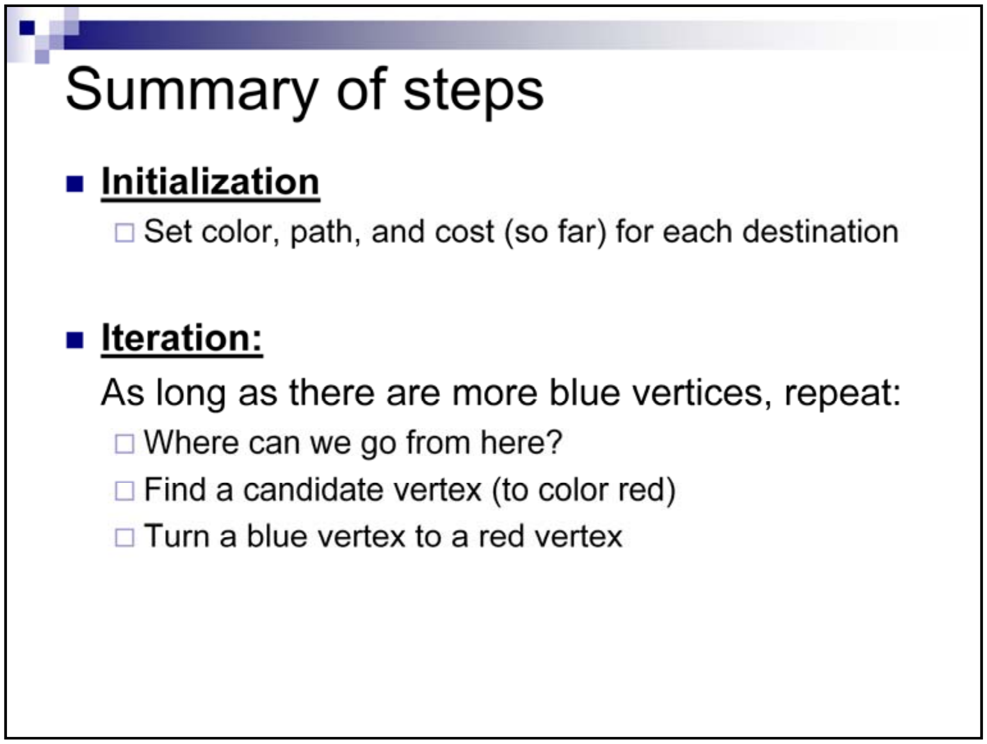
Make a Blue vertex Red



	$	Path
A	0	A
B	4	A→C→B
C	3	A→C
D	11	A→C→B→D
E	5	A→C→B→E

- Done with D; color it **Red**
- No other remaining (blue) vertices; we are DONE!

And we are done!

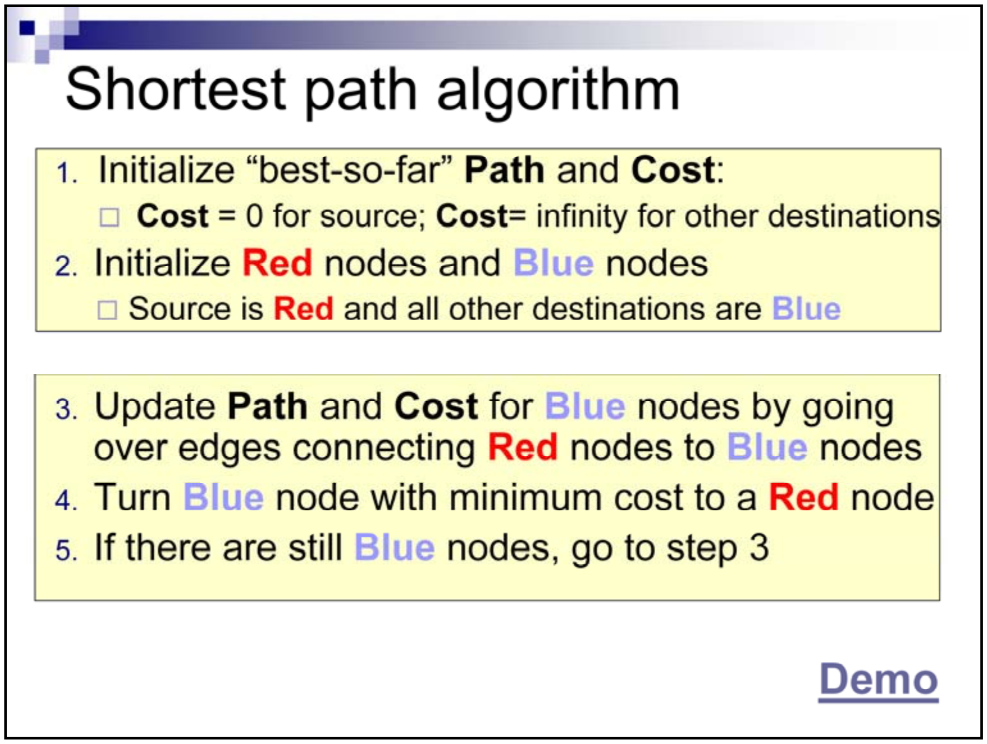


Summary of steps

- **Initialization**
  - Set color, path, and cost (so far) for each destination
- **Iteration:**

  As long as there are more blue vertices, repeat:
  - Where can we go from here?
  - Find a candidate vertex (to color red)
  - Turn a blue vertex to a red vertex

As summarized above, our process consisted of an initialization and then a bunch of iterations, each of which resulted in turning one of the blue vertices into red.



Shortest path algorithm

1. Initialize "best-so-far" **Path** and **Cost**:
   - **Cost** = 0 for source; **Cost**= infinity for other destinations
2. Initialize **Red** nodes and Blue nodes
   - Source is **Red** and all other destinations are Blue
3. Update **Path** and **Cost** for Blue nodes by going over edges connecting **Red** nodes to Blue nodes
4. Turn Blue node with minimum cost to a **Red** node
5. If there are still Blue nodes, go to step 3

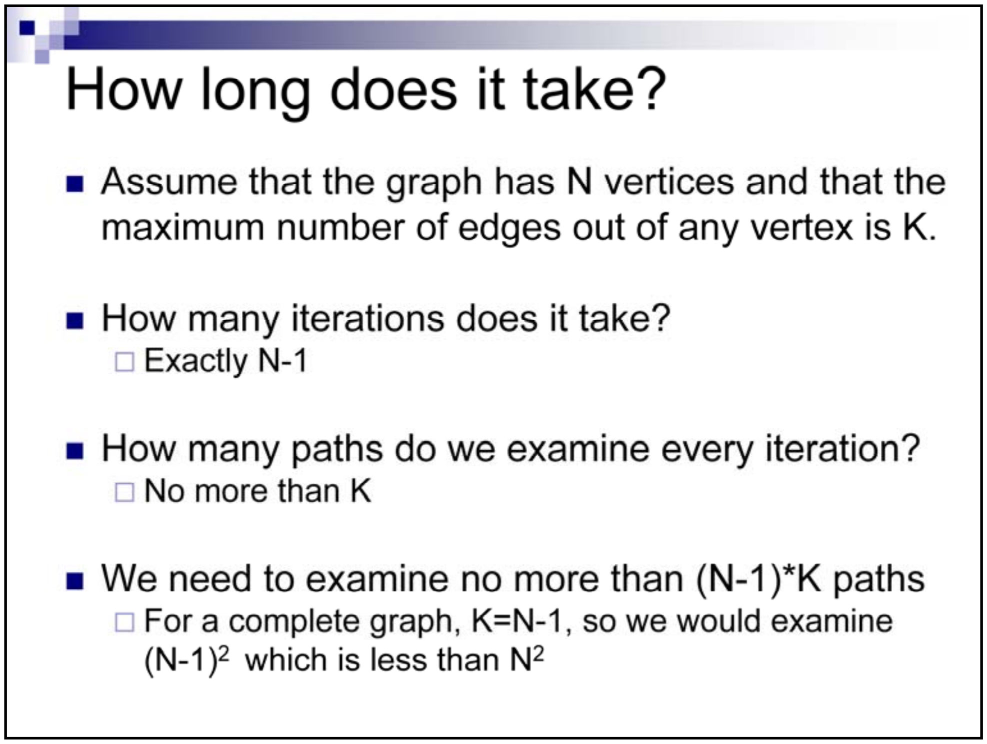
Demo

We can be more detailed in writing the procedure as shown above.

And, you can see the above procedure "in action" by trying the demo available at

http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html

(you need to have Java installed on your computer to be able to run this demo)



How long does it take?

- Assume that the graph has N vertices and that the maximum number of edges out of any vertex is K.
- How many iterations does it take?
  - Exactly N-1
- How many paths do we examine every iteration?
  - No more than K
- We need to examine no more than (N-1)*K paths
  - For a complete graph, K=N-1, so we would examine $(N-1)^2$ which is less than N^2

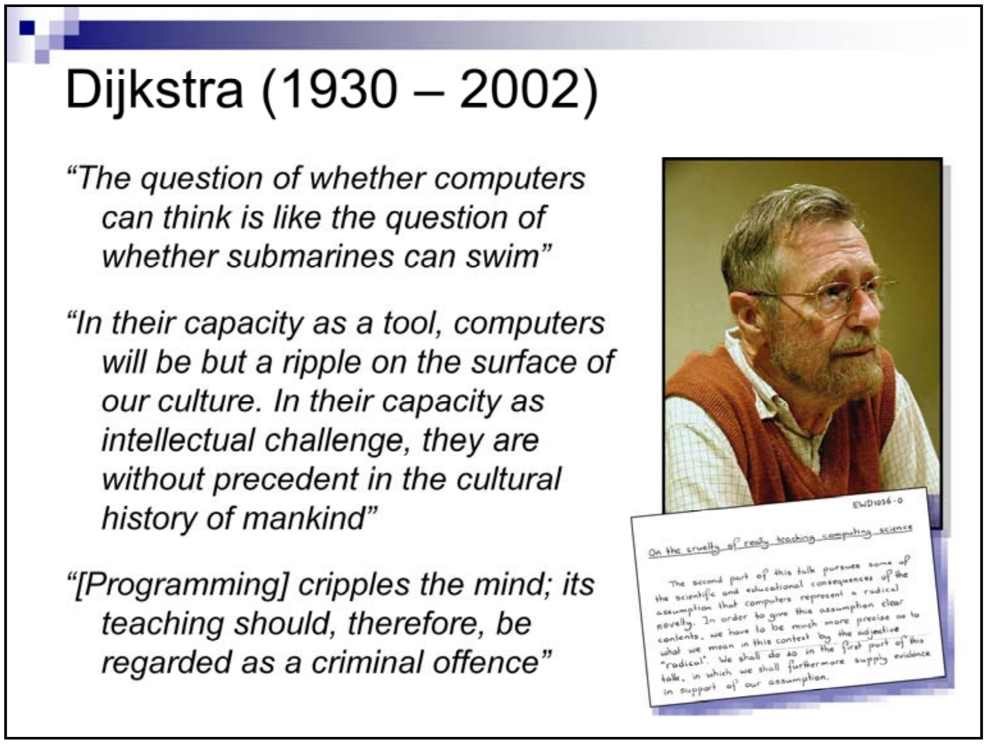
And, now, we ask the question that we always ask after we devise an algorithm – how good is this algorithm? How long would it take?

We assume that the graph on which we run the algorithm has N vertices and that the maximum number of edges out of any vertex in the graph is K. Note: The number of edges out of a vertex is called the degree of that vertex. So, K is the maximum degree of all vertices in the graph. Since there can be no more than N-1 edges out of a vertex, K cannot exceed N-1.

In each iteration of our algorithm, we examines some candidate paths. The set of candidate paths we examine in a given iteration are precisely those resulting from considering the edges out of the red node we have added in the previous iteration. At most we would have K such candidate paths (per iteration).

Our algorithm executes exactly N-1 iterations (recall that each iteration turns one of the vertices from blue to red, and since we started with N-1 blue vertices, we will execute exactly N-1 iterations).

So, the total number of paths we will evaluate will be no more than K*(N-1). And, since K is at most N-1, the number of such paths is no more than $(N-1)^2$ – i.e., quadratic in the number of vertices in the graph, which is far *far* better than what we had with the brute-force approach (as promised).

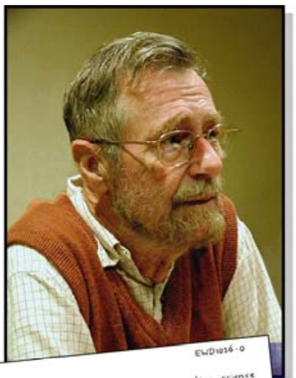


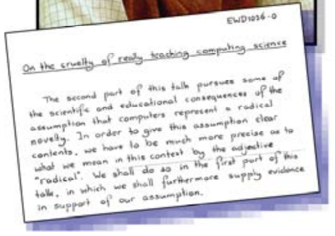
Dijkstra (1930 – 2002)

"The question of whether computers can think is like the question of whether submarines can swim"

"In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind"

"[Programming] cripples the mind; its teaching should, therefore, be regarded as a criminal offence"





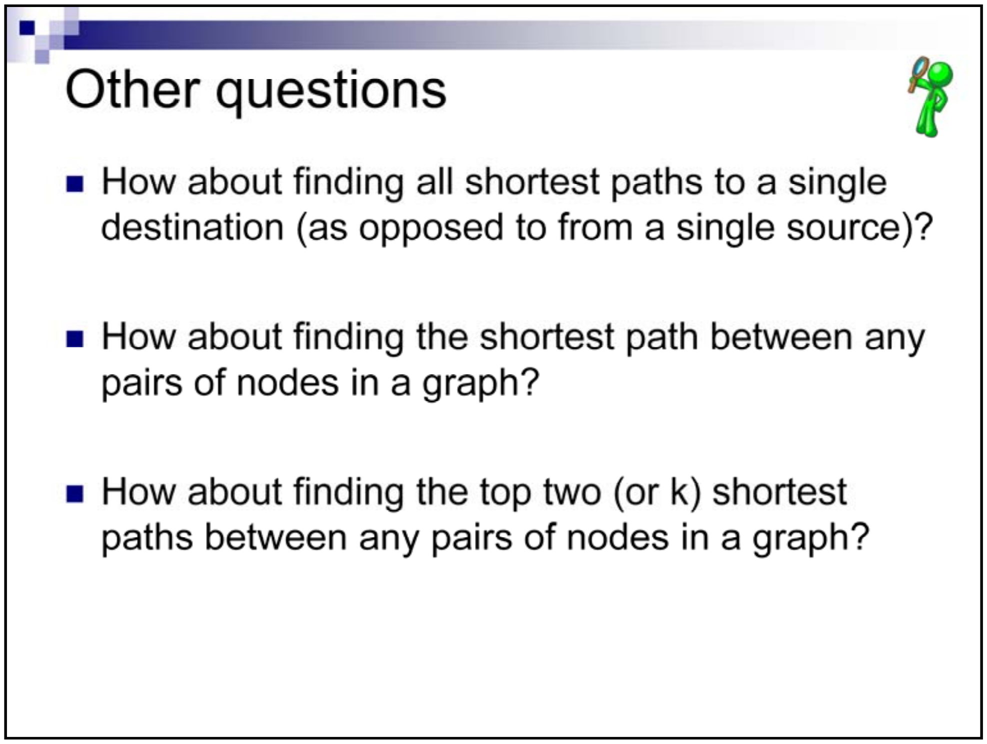
EWD1036-0

On the cruelty of really teaching computing science

The second part of this talk pursues some of the scientific and educational consequences of the assumption that computers represent a radical novelty. In order to give this assumption clear contents, we have to be much more precise as to what we mean in this context by the adjective "radical". We shall do so in the first part of this talk, in which we shall furthermore supply evidence in support of our assumption.

The algorithm we have just developed is due to a celebrated Dutch Computer Scientist by the name of Edsger Dijkstra who proposed it in 1959.

Appropriately, this algorithm is called "Dijkstra's Algorithm".



Other questions

- How about finding all shortest paths to a single destination (as opposed to from a single source)?
- How about finding the shortest path between any pairs of nodes in a graph?
- How about finding the top two (or k) shortest paths between any pairs of nodes in a graph?

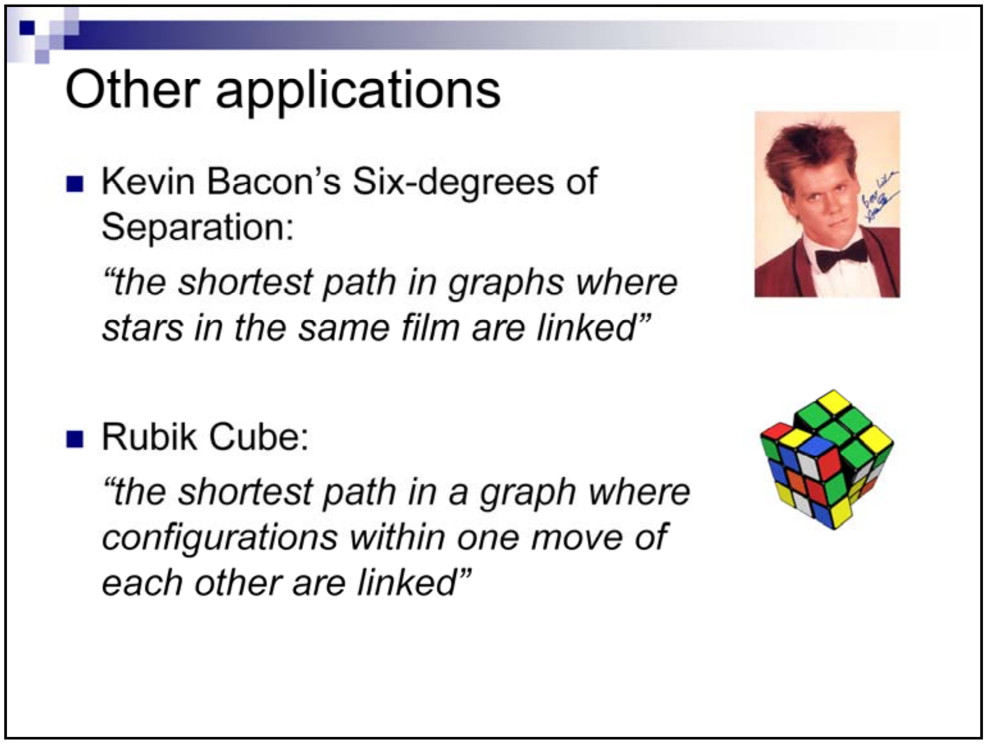
Can we use Dijkstra's algorithm to answer other similar questions to "finding the shortest path from one source to all destinations"?

Two direct extensions are to calculate the shortest path from all possible sources to a single destination, and finding the shortest path between any pairs of nodes.

We can do the former by applying Dijkstra's algorithm "backwards" – start from the destination and follow edges backwards to find the shortest paths from all sources.

We can do the latter by applying Dijkstra's algorithm from every possible starting vertex.

Interestingly, by introducing a slight generalization of the shortest path problem – namely find the two shortest paths (or in general k shortest paths) in a graph, one has to rethink the whole process (and the resulting algorithms become quite elaborate).



Other applications

- Kevin Bacon's Six-degrees of Separation:

  "the shortest path in graphs where stars in the same film are linked"

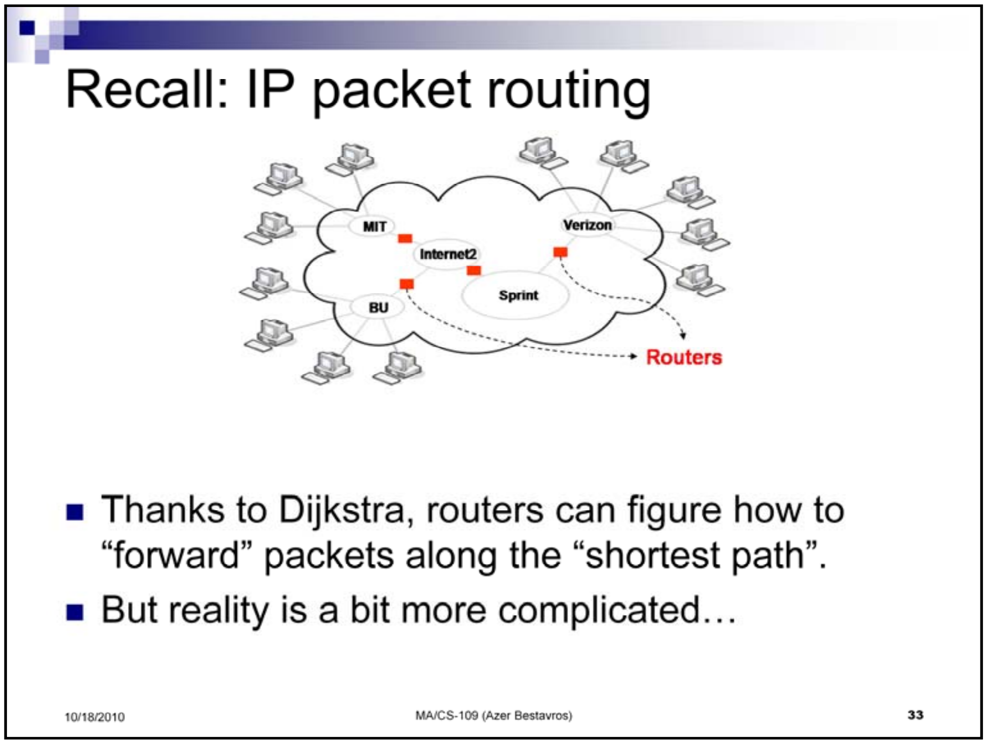
- Rubik Cube:

  "the shortest path in a graph where configurations within one move of each other are linked"

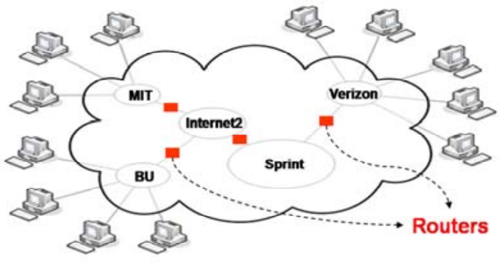
So, what does the computation of the shortest path allow us to do?

How about finding the least number of handshakes separating you from somebody else – a phenomenon that became popular by the "Six Degrees of Kevin Bacon" game whose goal is to link any actor to Kevin Bacon through the least number of connections (edges) where two actors are connected if they have appeared in a movie together. Turns out the shortest path from any actor to Kevin Bacon is around six – hence the name of the game (and phenomenon)!

In general, Dijkstra's shortest path algorithm can be used to search for the best way to go from one configuration to another through well defined changes in configurations. An example is the Rubik Cube.



Recall: IP packet routing



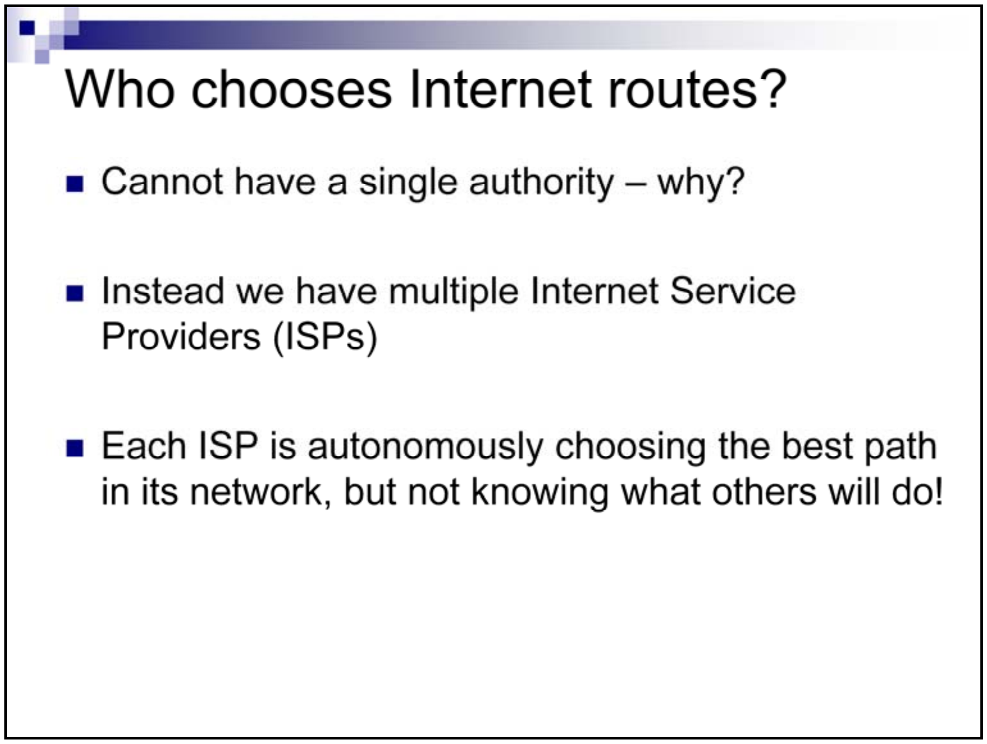
- Thanks to Dijkstra, routers can figure how to "forward" packets along the "shortest path".
- But reality is a bit more complicated…

10/18/2010 MA/CS-109 (Azer Bestavros) 33

Recall that routers are computers that link up **two** or more local networks. A router handles any packet whose destination is not local, storing it, and then forwarding it to the "next" network (hop) towards its destination.

Using Dijkstra's shortest path algorithm, we can figure out how to forward packets along the shortest path.

Unfortunately, reality (on the Internet) is much more complicated.



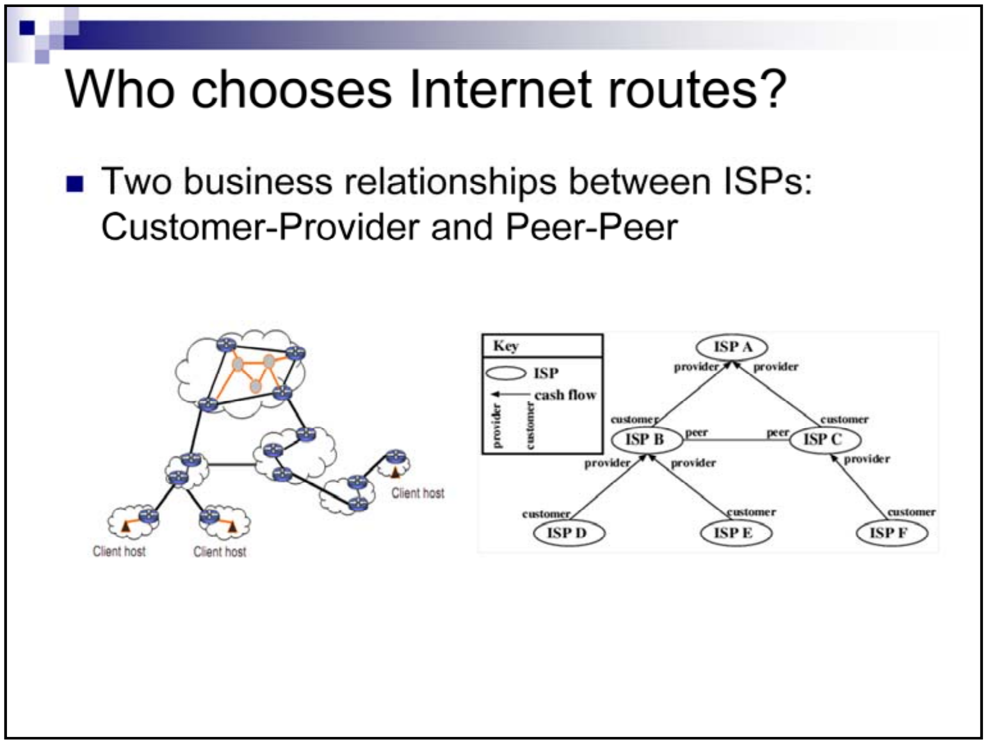
Who chooses Internet routes?

- Cannot have a single authority – why?
- Instead we have multiple Internet Service Providers (ISPs)
- Each ISP is autonomously choosing the best path in its network, but not knowing what others will do!

In order to compute the shortest paths, “somebody” must have access to the entire graph of the Internet. Having such an authority is not possible – nobody would be “trusted” to do this. Besides, the Internet is really an industry with many companies (called Internet Service Providers) making it up.

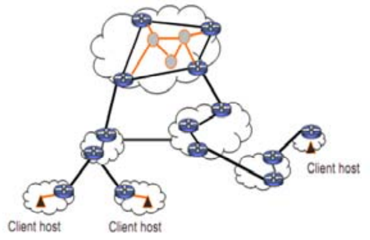
Within the confines of a single ISP network, we can run Dijkstra’s shortest path algorithm (or similar algorithms) to figure out the “best” paths within the ISP network.

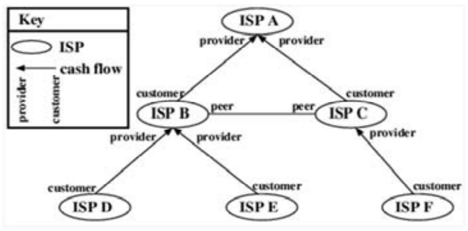
But, the problem is that Internet paths go over many ISPs, and each ISP makes its (routing) decisions independently and without knowledge of other ISPs’ (routing) decisions.



Who chooses Internet routes?

- Two business relationships between ISPs: Customer-Provider and Peer-Peer



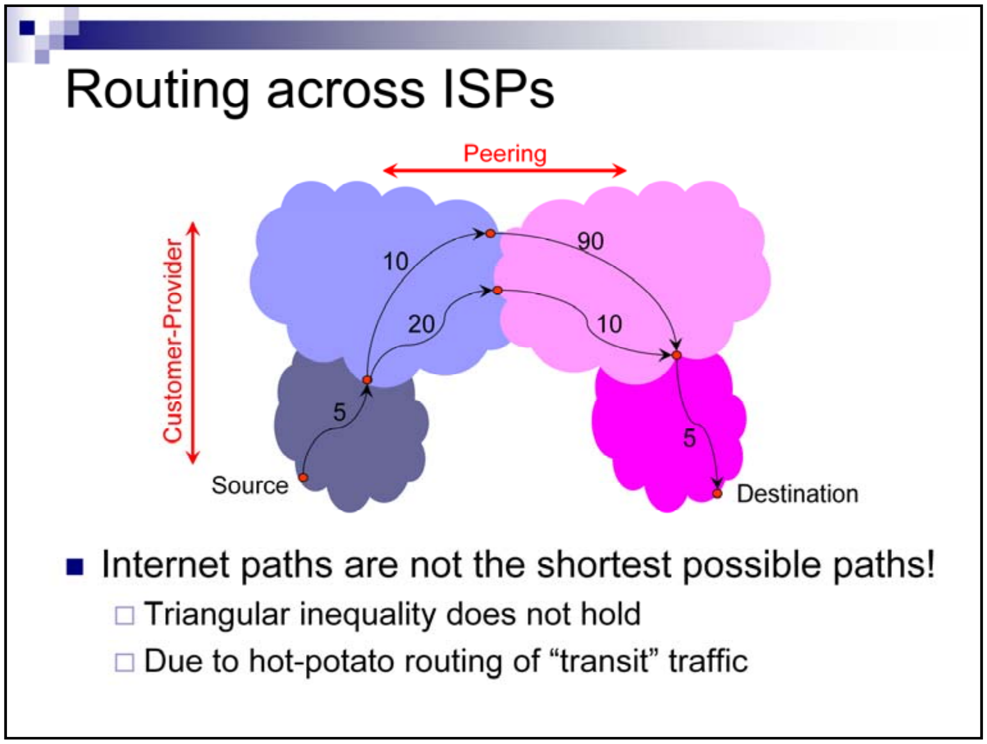


So, how do ISPs relate to one another?

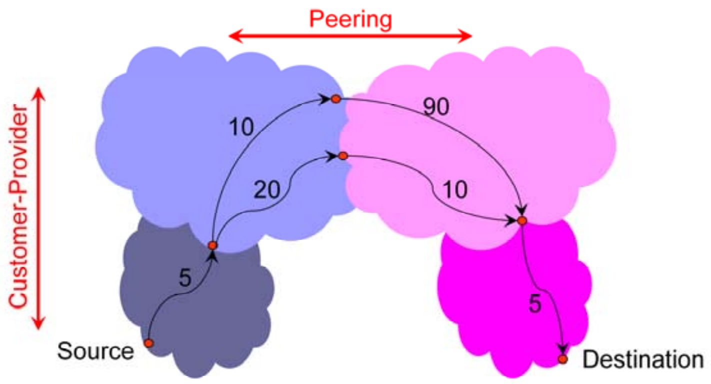
Basically, there are two possible business relationships between ISPs: It is either a "customer-provider" relationship where one ISP is a customer and the other is the provider (i.e., the customer gets internet connectivity by paying a fee to the provider), or else it is a "peer-to-peer" relationship (i.e., the two ISPs agree to exchange traffic without having to pay each other any fees). Typically, these peering relationships exist between the larger ISPs (e.g., AT&T peering up with Sprint).

In a customer-provider relationship, all Internet traffic (packets) from/to the customer is handled by a single ISP through a single router. So, when a packet must be forwarded from a customer to a provider, there is only one possibility for the packet to cross from the customer network to the provider network.

In a peer-to-peer relationship, there may be multiple routers on the border between the peering ISPs, so when a packet must be forwarded from one peer to another, it may have multiple possible crossing points.



Routing across ISPs



- Internet paths are not the shortest possible paths!
  - Triangular inequality does not hold
  - Due to hot-potato routing of "transit" traffic

This situation is illustrated above.

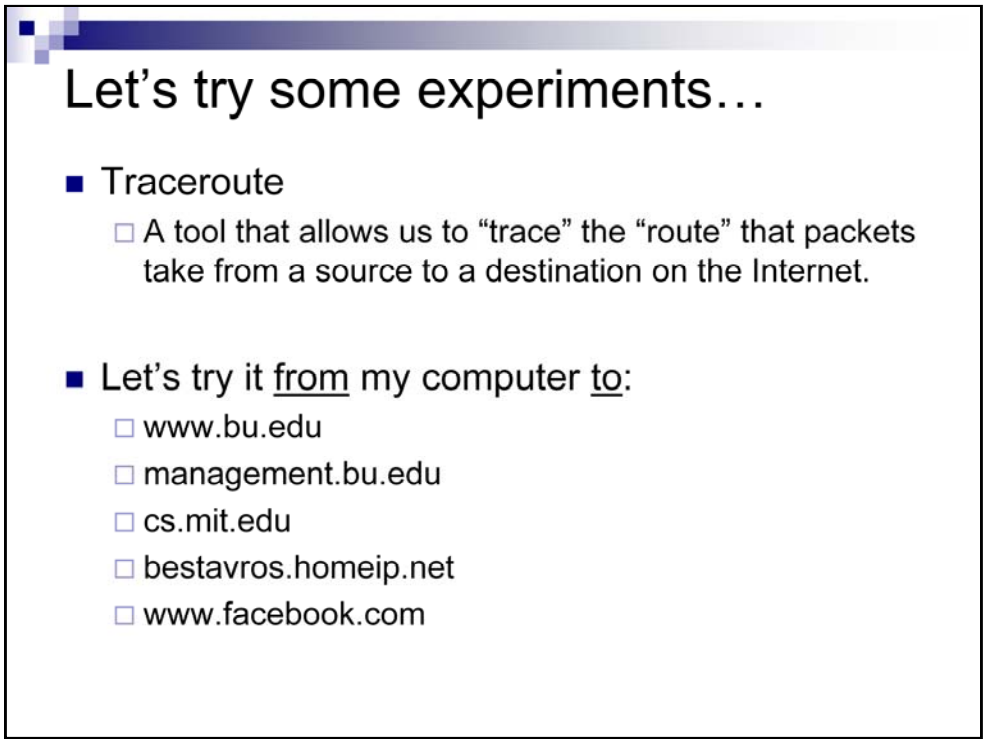
Since each ISP makes routing (e.g., shortest path) decisions without knowledge of the costs/decisions at other ISPs, one can see that (in the above example) the route selected from the source to the destination will not be the shortest path!

This phenomenon (due to peer-to-peer ISPs dumping transit traffic out of their networks as soon as possible) is called "hot-potato" routing.

There are other complications resulting from the autonomous nature of ISPs – including the possibility of routes over the Internet to become unstable (so a packet may end up going around in loops, for example), requiring us to add yet more functionalities to the Internet .

As we mentioned in earlier lectures, each functionality on its own is simple enough and well modeled and understood (and thus "beautiful"). Shortest path routing is an example of such a beautiful functionality.

Yet, by adding layers upon layers of functionality we end up with what may appear to be a "beast"!



Let's try some experiments…

- Traceroute
  - A tool that allows us to "trace" the "route" that packets take from a source to a destination on the Internet.
- Let's try it <u>from</u> my computer <u>to</u>:
  - www.bu.edu
  - management.bu.edu
  - cs.mit.edu
  - bestavros.homeip.net
  - www.facebook.com

One can check the paths that packets take on the Internet using tools that tell us the various routers that handle a packet on its way from a source to a destination.

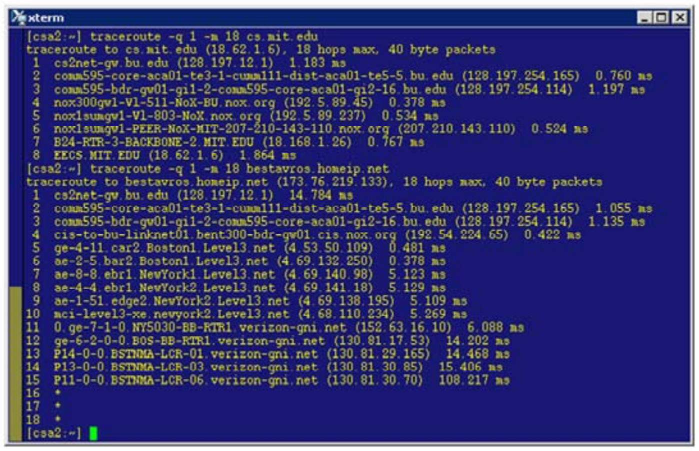
One such tool is called "traceroute" (or "tracert" on windows machines). More information on traceroute can be found on the web, e.g.:
http://www.exit109.com/~jeremy/news/providers/traceroute.html

One can try these tools from web sites that allow you to trace the path that packets take from such web sites to any arbitrary computer on the Internet. Here is an example web site you can play with (courtesy of Princeton U):
http://www.net.princeton.edu/traceroute.html

A directory of other places where one could try traceroute is available through the following web site:
http://www.traceroute.org/

And here are some example web sites that provide "visualizations" of the results one gets from traceroute:
http://www.yougetsignal.com/tools/visual-tracert/
http://visualroute.visualware.com/

Tracerouting Internet paths from BU



```
[csa2:~] traceroute -q 1 -m 18 cs.mit.edu
traceroute to cs.mit.edu (18.62.1.6), 18 hops max, 40 byte packets
 1  cs2net-gw.bu.edu (128.197.12.1)  1.183 ms
 2  comm595-core-aca01-te3-1-cumm111-dist-aca01-te5-5.bu.edu (128.197.254.165)  0.760 ms
 3  comm595-bdr-gw01-gi1-2-comm595-core-aca01-gi2-16.bu.edu (128.197.254.114)  1.197 ms
 4  nox300gw1-V1-511-NoX-BU.nox.org (192.5.89.45)  0.378 ms
 5  nox1sumgw1-V1-803-NoX.nox.org (192.5.89.237)  0.534 ms
 6  nox1sumgw1-PEER-NoX-MIT-207-210-143-110.nox.org (207.210.143.110)  0.524 ms
 7  B24-RTR-3-BACKBONE-2.MIT.EDU (18.168.1.26)  0.767 ms
 8  EECS.MIT.EDU (18.62.1.6)  1.864 ms
[csa2:~] traceroute -q 1 -m 18 bestavros.homeip.net
traceroute to bestavros.homeip.net (173.76.219.133), 18 hops max, 40 byte packets
 1  cs2net-gw.bu.edu (128.197.12.1)  14.784 ms
 2  comm595-core-aca01-te3-1-cumm111-dist-aca01-te5-5.bu.edu (128.197.254.165)  1.055 ms
 3  comm595-bdr-gw01-gi1-2-comm595-core-aca01-gi2-16.bu.edu (128.197.254.114)  1.135 ms
 4  cis-to-bu-linknet01.bent300-bdr-gw01.cis.nox.org (192.54.224.65)  0.422 ms
 5  ge-4-11.car2.Boston1.Level3.net (4.53.50.109)  0.481 ms
 6  ae-2-5.bar2.Boston1.Level3.net (4.69.132.250)  0.378 ms
 7  ae-8-8.ebr1.NewYork1.Level3.net (4.69.140.98)  5.123 ms
 8  ae-4-4.ebr1.NewYork2.Level3.net (4.69.141.18)  5.129 ms
 9  ae-1-51.edge2.NewYork2.Level3.net (4.69.138.195)  5.109 ms
10  mci-level3-xe.newyork2.Level3.net (4.68.110.234)  5.269 ms
11  0.ge-7-1-0.NY5030-BB-RTR1.verizon-gni.net (152.63.16.10)  6.088 ms
12  ge-6-2-0-0.BOS-BB-RTR1.verizon-gni.net (130.81.17.53)  14.202 ms
13  P14-0-0.BSTNMA-LCR-01.verizon-gni.net (130.81.29.165)  14.468 ms
14  P13-0-0.BSTNMA-LCR-03.verizon-gni.net (130.81.30.85)  15.406 ms
15  P11-0-0.BSTNMA-LCR-06.verizon-gni.net (130.81.30.70)  108.217 ms
16  *
17  *
18  *
[csa2:~]
```

Live demo of tracerouting from my computer to some other computers on the Internet (close by, e.g., at BU, or far away).

Interestingly, for a packet to make it from BU to my home in the western suburbs of Boston, it had to go all the way to New York and back!!

We will have more to say about (and opportunities for running experiments like) this in future lectures.