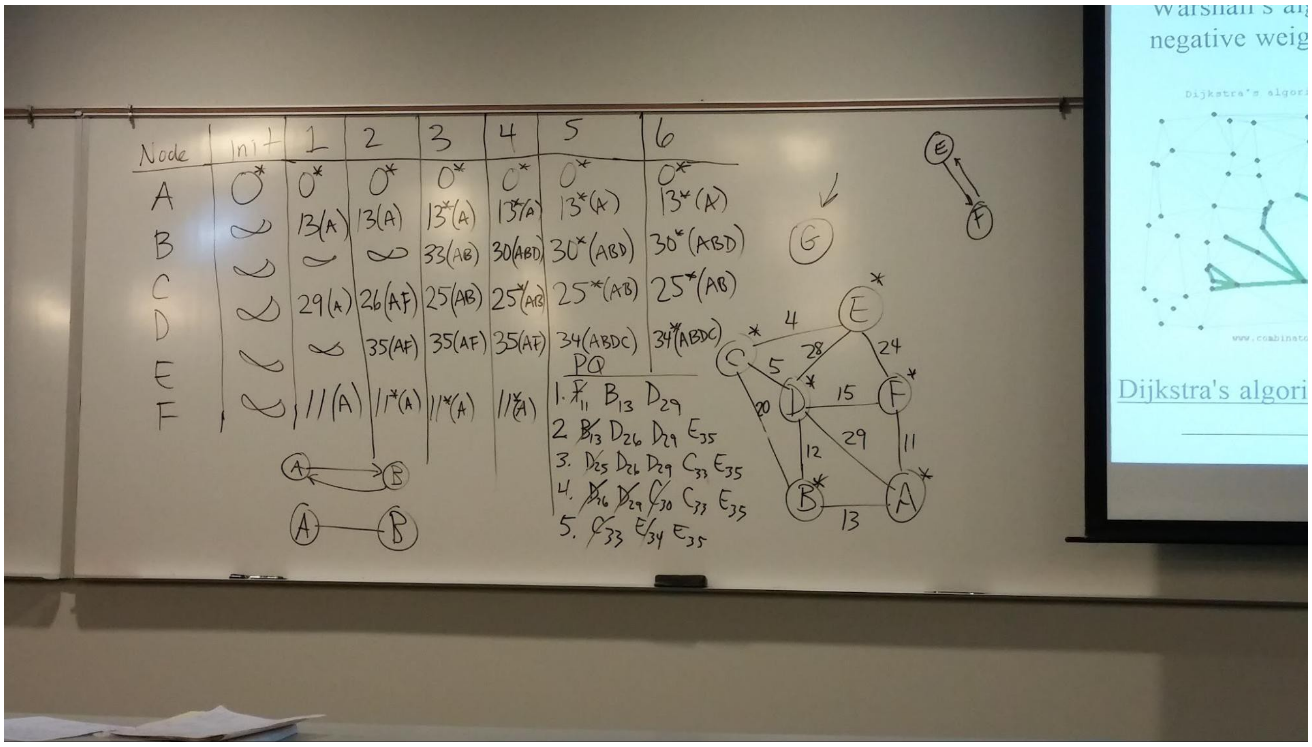
CS280 Notes 3/28

By Christian Sagel

We want to find the cheapest way to get from A to B.
Pseudocode for Djkstras Algorithm: Given a source node, we can find the shortest distance to every other node in a graph/

- Choose a node to be the source or starting point.
- Initialize source to 0 cost and mark as evaluated.
- Initialize all nodes to infinite cost from the source.

For each node, y, adjacent to source.



Node	Init	1	2	3	4	5	6
A	0*	0*	0*	0*	0*	0*	0*
B	∞	13(A)	13(A)	13*(A)	13*(A)	13*(A)	13*(A)
C	∞	∞	∞	33(AB)	30(ABD)	30*(ABD)	30*(ABD)
D	∞	29(A)	26(AF)	25(AB)	25*(AB)	25*(AB)	25*(AB)
E	∞	∞	35(AF)	35(AF)	35(AF)	34(ABDC)	34*(ABDC)
F	∞	11(A)	11*(A)	11*(A)	11*(A)		

PQ

1. ~~F_{11}~~ B_{13} D_{29}
2. ~~B_{13}~~ D_{26} D_{29} E_{35}
3. ~~D_{25}~~ D_{26} D_{29} C_{33} E_{35}
4. ~~D_{26}~~ ~~D_{29}~~ ~~C_{30}~~ C_{33} E_{35}
5. ~~C_{33}~~ ~~E_{34}~~ E_{35}

We only have the adjacent list to know its neighbors. We will always choose the cheapest edge to follow (greedy algorithm). From there we will see these nodes. We will keep them on a priority queue.
Evaluated means we have found the cheapest route.
We assume that infinity, practically we can cast -1 to unsigned giving you the largest.
In parenthesis we put what node it took us to get there.

Node	Init	1	2	3	4	5	6
A	0* (Evaluated)	0* (Evaluated)	0*	0*	0*	0*	0*
B	inf	13(A)	13(A)	13(*A)	13*(A)	13*(A)	13*(A)
C	inf	inf		33(AB)	30(ABD)	30*(ABD)	30*(ABD)
D	inf	29(A)	26(AF)	25(AB)	25*(AB)	25*(AB)	25*(AB)
E	inf	inf	35(AF)	35(AF)	35(AF)	34(ABDC)	34*(ABDC)
F	inf	11(A)	11*(A)	11*(A)	11*(A)	11*(A)	11*(A)

We are looking for the cheapest route for *every node starting from the source!*
Don't be alarmed when you see multiple copies of your nodes as you step in Visual Studio, as that's how priority queues work.
On every step, we evaluate another node.

1. Start with A. It has 3 neighbors, the edge AF costs 11, AB 13, AD 29.
   PQ: F(11), B(13), D(29)
2. We will now pull F from the PQ, mark it as evaluated. F can go E or D. D costs 15. Going from A to F costs 11, then F to D 15. We will now update D to 26.
   PQ: B(13), D(26), D(29), E(35)
3. We now evaluate B.
   PQ: D(25), D(26), D(29), C(33), E(35)
4. We evaluate D. D has 5 neighbors. 3 of them are already evaluated. D goes to C, costing 5. Going from A to C through ABDC will cost 30. D also has the neighbor E. Currently it costs 35, which is less than ADE
   PQ: D(26), D(29), C(30), C(33), E(35)
5. We now evaluate C. To get to E from C only costs 4, so we update.
   PQ: C(33), E(34), E(35)
6. We evaluate E. All its neighbors have been evaluated so we do nothing.

At the end:

TO	PATH	COST
A	A	0
B	A-B	13
C	A-B-D-C	30
D	A-B-D	25
E	A-B-D-C-E	
F	A-F	

We should have a node's class, edge class. Also and adjacency list. Internally we can do whatever want, but when we communicate with the driver we have to put.
If we have an edge class and overload the less than operator it will work beautifully when.
If there’s no route, we return -1. (Imagine an unconnected node)
We will have both directed and undirected graphs.
If we have a tie, we will choose the the edge with the lowest value (tiebreaker). First base it on the weight of the edge, and if there's a tie base it on the ID of the node.

Next assignment due on Sunday, April 10.
ALGraph.
It has a small interface. Constructor with the number of nodes to have. If he tells us we are gonna get 10 nodes, we will definitely get them so we don’t have to check them.
We have Directed and Undirected edges.
Our Undirected Edge is just going to call Directed Edge. Because they are all going to be directed graphs. This way we don’t have to distinguish between directed and undirected internally.

We decide how to store the edges.
We are going to need dijsktra’s algorithm before we do the extra credit, *SearchFrom*.
ALISt used for debugging.
TestDjkistra0 is the test based from the in-class example.

Graphviz, part of doxygen. When you have hierarchies in C++, it will draw them for you.
Mead modified the driver with Graphviz. It can draw lots of data structures.
Directed ->, Undirected -

He is giving out the pseudocode for the extra credit too.
We can’t check whether we are doing breadth/depth first. We will make it templated to work with one function. We will derive from std::stack so they have the same interface.
He will give us a starting node, an unsigned index.

For example, when calling UDEdge method with edge AB, will just call DEdge and pass in AB and BA.

Notes: Djkstra’s algorithm only works for positive weights. Runtime complexity O(m log n) where m is the number of edges and n is the number of nodes.
Thankfully Priority Queues are implemented on the heap.
If you wanna build a sorted linked list, you can build it as you go or throw them to the back and sort them later. (Sorting then later is more expensive)

We may store all our edges in an std::vector calling push_back. If we have 1000 edges, tat will be very fast. A set will keep it sorted.

He finds building the graph slower than running the algorithm if keeps it sorted.
You have to decide where you want to pay the cost, in building the graph or running the algorithm.
We don’t have to sort our priority queue, as that’s taken care of for us.
The path can be on a vector or list.
Auxiliary data structures matter.
Set is a red-black tree.
Graph algorithms are a big topic.
Djkstra’s algorithm is a nice beginner algorithms. A* is another algorithm which uses heuristics.

Mead has a lot of experience writing GUIs.
Mead is getting the std cout output from the process by using pipes.
The compiler is a text-based console app. When visual studio launches the compiler it sets up a pipe between the compiler and the IDE, so it gets all the output.