Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

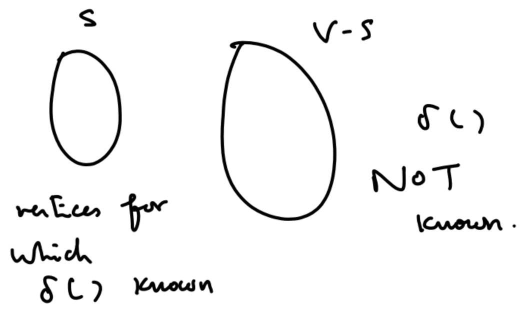
Department: Computer Science and Engineering

Institute: IISc

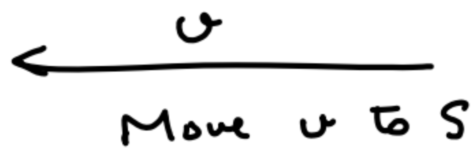
Week: 04

Lecture 17 Dijkstra Algorithm P2

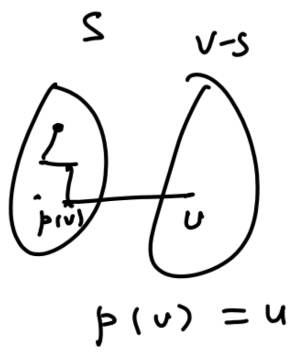
Namaskara we continue our discussions on Dijkstra's algorithm. In Dijkstra's algorithm we solve the single source shortest path problem by determining the shortest path weight in incremental order. So, we will do n minus 1 iterations and find the shortest path weights of n minus 1 vertices in each iteration we find one vertex for which the shortest path weight is determined. So, we will maintain 2 sets S and another set V minus S, so S is the set of all vertices for which delta value are known, this is not, delta value are not known.



So, we will work with the vertices in V minus S, identify a vertex for which the delta is known then move that to so determine a vertex v in V minus S such that delta v is known. or delta v is computed since for v, delta v is known, we have to move v to S okay this is what we repeatedly do.



So how do we determine the vertex in V minus S, we keep for every vertex in, for every v belonging to V minus S, we maintain 2 pieces of information. The first one is d of v which is the weight of shortest special path, what we mean by special path is a path from S to V with all vertices other than v are in S okay. So, if this is S there may be several vertices and then this is v, is a special path okay. So the special path ends with a crossing edge, crossing edge is an edge going from S to V minus S and this is previous of v, this is the last edge so we always say the last edge in the form of the vertex and it is a previous vertex so pv equal to u where uv is the last edge of the shortest special path.



So, these two pieces of information we maintain use this to determine the v. So, how do we determine v we have seen in the previous session that if v is a vertex in V minus S such that dv is less than or equal to du for all u belonging to V minus S

$$d[v] \leq d[u] \; \forall \, u \; \in V - S$$

That is v is a vertex with minimum d value, v is a vertex with minimum d value. Every vertex has got a d value associated with it. The value associated with these minimum among all the vertices in V minus S okay. In this case we have shown that let me write, then dv is in fact delta v

$$d[v] = \delta(v)$$

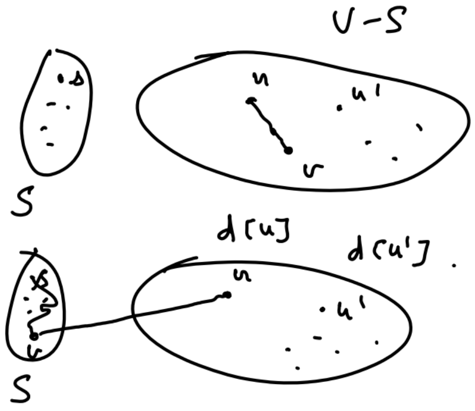
That means this implies delta v is known for v which implies we move v to S. Now what is to be done? Look at the definition of the special path, definition of a special path is that it is a path which is going through only the vertices of S and then reach. Now S has become larger, we have one more vertex added to S, therefore d values are to be updated okay. Since S has been updated to S union v you have moved we need to update du values for u in V minus S okay.

$$S \cup \{v\}$$

$$u \; \in V - S$$

The following observation is helpful in identifying for what vertices we have to do the updating and how we do the updating. Okay, so it is a very common sense and simple observation once this is done we are done with the details of the algorithm. We will put all the things together and then the final version of the algorithm can be drafted. So what has happened now, let us draw two pictures this is S and this is V minus S and there are several vertices u is here there is a v and then we have moved v over here. So s and v has been moved here okay this is small s this is the set S.

Now you take Let us consider 2 vertices one is u another is u dash okay, so assume that you have u adjacent to v and u dash is not adjacent to v okay, so let us draw u in a more comfortable way this is V minus S this is v this is u this is u dash and there are other vertices v to u you have an edge and you have moved v. this is u and this is u dash and there are other vertices. So after moving v this is the scenario okay. Now d of u and d of u dash what is the meaning of d of u dash it is a minimum weight among all these special pathweights. By moving v no new special paths are created for u dash the same set of special paths continue to be there here. Because adding v is not creating any special path for u dash okay. However for u a new special path is created that is because v has come here now s to v you have a path special path you have moved v also here so that path has become a special path to u now okay.



Therefore now I have to update d of u I do not have to worry about d of u dash I do not have to worry about d of u dash but I have to examine d of u because for u some new special paths have come into picture, I have a minimum among the old set of special paths. A new special path has come maybe this is a shorter than the other one, how do we know what is the length of or what is the weight of this new special path it is dv because this is the path and plus weight of vu, dv plus weight of u

$$d[v] + w[v, u]$$

This is the weight of new special path to u this is the weight of a new special path u whether this new special path is updating the minimum, how do we check, if d of u is greater than d of v plus weight of uv then the new special path is the minimum special path it has got a smaller weight

$$d[u] > d[v] + w[u, v]$$

Then d of u is updated to d of v. plus wuv and now this is the updated shortest special path the previous vertex so previous of u equal to v.

$$d[u] = d[v] + w[u, v]$$

$$p[u] = v$$

So you can see that this is the previous vertex to u in this picture u is attached with v earlier somebody else some other path was defining the minimum and that might be the previous vertex. Now this path is defining the minimum special path or shortest special path therefore the previous of v is u, right. This I have to do only for neighbors of v. if a vertex is not even a neighbor of v I do not have to worry about them for those vertices the du dash value remains same du dash where u dash does not belong to adjacency of v does not change. and if u is in the adjacency of v it may change if this is the condition, if it is defining a new therefore what is the updating code. for all u in adjacency of v only for this vertices I will consider for every vertex adjacent to v that is in V minus S so for all u in if u belongs to V minus S and this condition d of u is greater than d of v plus weight of v u.

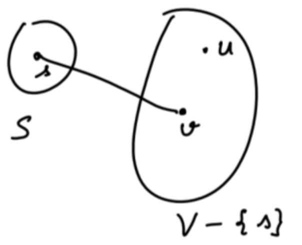
$$u \in V - S \text{ and } d[u] > d[v] + w[u, v]$$

If this is the case then d of u is equal to d of v plus weight of v u and p of u equal to v.

$$d[u] = d[v] + w[u, v]$$

$$p[u] = v$$

That is all is the updating to be done. So let us see all these things are done assuming a d value is available. So how will the d values look like initially we have to initiate the d values properly. Let us start with the following simple initial configuration. Okay. we will have in S only the vertex s and then here all the vertices V minus s okay consider a vertex v here.



What is a special path to v? Special path is starting from s it will go through all the vertices of S and then with a crossing edge it will reach v that is a special path right. There is no other vertex in S therefore the special path is nothing but the edge there is only one special path. its weight is so the edge sv is special path to v from s this is the only one. So weight of the minimum weight of special path is d of v is nothing but weight of sv.

$$d[v] = w[s, v]$$

Suppose there is a vertex u that is not adjacent to s from s to u there may be paths but there are no special paths because there is no edge connecting s to u because special path means all vertices must be in this set, to u from s there may be paths but there are no special paths. u does not belong to adjacency of s then no special path exists at this point from s because S has no other vertices.

Therefore special path means it should be an edge when a special path does not exist we always set infinity okay

$$d[v] = \infty$$

Therefore this is the initial configuration. The initial configuration is d of v equal to weight of sv if v belong to adjacency of s this is equal to infinity if the vertex is not in adjacency of s okay I will write it cleanly. is equal to infinity if v does not belong to adjacency of s. This is the way you initialize and d of s is 0,

$$d[v] = w[s, v] \text{ if } v \in \text{Adj } (s)$$

$$= \infty \text{ if } v \in \text{Adj } (s)$$

$$d[s] = 0$$

So d array is initialized like this and what about the previous array. previous of s not only s okay previous of s is undefined, previous of v equal to s if

v belongs to adjacency of s because you have a special path okay look at this situation. previous of v is s in this picture you can see that previous of v is s. Previous of u if it is not adjacent it is not defined there is no path and is undefined if v does not belong to adjacency of s.

$$p[s] = \perp \text{ (undefined)}$$

$$p[v] = s \text{ if } v \in \text{Adj } (s)$$

$$= \perp \text{ if } v \notin \text{Adj } (s)$$

So this is how you initialize d array and P array okay. How are you going to represent the partition the split S, V minus S because we have to often check and we have to move from one set to another okay we represent S V minus S by a Boolean array. Let us say in

S, In Sv equal to 1 if v is in S because In Sv equal to 0, this is 0 if v is not in S okay it is a boolean array it is a flag indicating it is in S or not. okay.

$\text{In} - S[v] = 1 \text{ if } v \in S$

$\text{In} - S[v] = 0 \text{ if } v \notin S$

Therefore this implies, In S of v equal to 0 for v in V minus S

$\text{In} - S[v] = 0 \text{ for } v \in V - S$

Because V minus S means these are the sets that are not in S therefore, if you want to check whether vertex is in V minus S you have to look into this Boolean array you just see whether In S of v is 0 if that is the case that vertex is in V minus S. Okay, so how do you move a vertex from V minus S to S? How do you move? How do you implement that movement? Okay, in picture we are drawing around and other things but how computationally the computer program has to do it mathematically, so it is very simple. If it is in V minus S, In SV will be 0, okay. So set in SV equal to 1, so earlier it will be 0 because it was in V minus S. Now it has been moved means it is in S.

How do you know that? How will the program know that a vertex is in S? This Boolean array value will be set to 1. That is all. This implements moving of a vertex from V minus S to S, okay. So, Dijkshtra G equal to V E w s given this as an input it has to produce for each V the delta v okay the delta v will be available in an array called d okay. in the end dv will be equal to delta v as output and pv is the previous vertex to v in the shortest path in other words pvv is a Bellman edge okay.

$$d[v] = \delta[v]$$

Initialize d array and p array. We have seen how that is done okay.

$$d[\quad] \text{ and } p[\quad], S, V - S$$

Two find a vertex v in V minus S such that d of v is less than or equal to d of u for all u in V minus S.

$$d[v] \leq d[u] \;\forall\, u \in V - S$$

So for each vertex in V minus S you look at the d value and find the minimum there will be some vertex for which the minimum value so by scanning the d array only for the elements of V minus S we can find the vertex v with minimum d value, third step is move V to S. move V to S okay. You have to initialize d, p and here itself let me specify S, V minus S.

You have to initialize S V minus S how do you initialize S V minus S you put only the source vertex s the source which is a part of the input put that in S all other vertices in V minus S okay but how do we actually do that we have a Boolean array for that. So let us see how this is okay let me see whether I have space in the previous page okay. So initializing, initialize S V minus S as the singleton s and V minus s. In S of s equal to 1 because this one is included in S and In S of v equal to 0 if v is not equal to s for all other vertices. you set In Sv equal to 0 for s you set In S s as 1 this represents the partition, this represents the initial partition.

$$(S, V - S) \to \{s\}, \{V - \{s\}\}$$

$$\text{In} - S[s] = 1$$

$$\text{In} - S[v] = 0 \text{ if } v \neq s$$

For the initial partition you know how to initialize d you know how to initialize p these are all the ways in which you have to initialize d and v for okay. So initialize d, p, s, V minus S after that find a vertex in v such that d of v is less than or equal to move v to s we have seen how moving of v is to be implemented. The fourth and final step is update d value and p value for every u in V minus S we have to update this. Although in the pseudo code I have written update d of u and p of u for every vertex you do not have to really touch those vertices that are not adjacent to v because only v has moved to s. So this step is expanded like this we expand step 4 as follows okay.

If u belongs to adjacency of v and d of u is greater than d of v plus weight of v u then you update d of u d of u equal to d of v plus weight of vu and p of u equal to v.

$$d[u] = d[v] + w[u, v]$$

$$p[u] = v$$

Therefore, fourth step is executed only for the neighbors of. V adjacency of V okay. So this completes the specification of Dijkstra 's algorithm the complete specification of the Dijkstra's algorithm is here we have put. all the ideas that we have developed in bits and pieces and integrated them and given in this pseudo code.

So this is what you have to do after initializing you have to do 2, 3 and 4 repeatedly. So, we have to do 2, 3 and 4 repeatedly that is not mentioned here I will mention now. So, basically we will do like this, I will create a space here, this is a single phase.

One initialize the partition S V minus S based on this the d array and p array then while S is not equal to V,v is the vertex set while S is not equal to V you repeat 2, 3 and 4 okay, so you can perhaps call this as step 2 it is easy to visualize that way. you call this while loop as step 2 and finding this vertex as within the while loop 2a moving this as 2b because this is all in the while loop and update d is 2c we expand step 2c as follows okay just we are fitting in 2c as follows.

So this will be in the while loop after the while loop the third step return d array and p array. okay so we will write like this return d of v and p of v for all v belonging to V this output that we want comment d of v is in fact delta v and pvv is the Bellman edge.

Complete pseudocode for Dijkstra's Algorithm

```
Dijkstra $(G(v, E, w, s))$
    1. Initialize $(S, V - S), d[\quad], p[\quad]$
    2. While $S \neq V$
        2.a find a vertex $v$ in $V - S$
        such that $d[v] \leq d[u] \forall u \in V - S$.
        2.b move $v$ to $S$
        2.c update $d[u]\ p[u]$ for each $u \in V - S$
If $u \in Adj\ [v]$ and $(d[u] > d[v] + w(v, u))$
Then
        $d[u] = d[v] + w(v, u)$
        $p(u) = v$
  return $(d[v],\ p[v],\ \forall v \in V)$
```

So this you can use to actually construct the shortest path okay. So this completes the description of Dijkstra 's algorithm. We will now take a quick look at the complexity of this method okay.

S initially had only one vertex and you are moving one vertex in each iteration therefore the while loop of step 2 will be executed n minus 1 times, what is n cardinality of v equal to n initially S has got only one vertex s and in each iteration of the while loop you are adding one vertex to S. you are finishing when s equal to v that is when all vertices are included in v that is for all vertices when I know the shortest path weight I complete. So n minus 1 vertices are to be added therefore n minus 1 iterations are to be done therefore while loop will be executed n minus 1 times okay. In each time you are finding the minimum element in 2a what is the cost of 2a you have to go through all vertices in V minus S and find the minimum, initially V minus S has got n minus 1 elements then n minus 2 elements then n minus 3 elements and so on. total sum of cost of 2a across all iterations is n minus 1 plus n minus 2 which is order n square.

$$(n-1)+(n-2)+\cdots+1=O(n^2)$$

Total sum of or total cost of 2b across all iterations is n minus 1 because in each iteration look at 2b, 2b is moving V to S, moving V to S is done by setting the Boolean flag to 1 that is one step right. Therefore, 2b when you implement n minus 1 times the total cost will be n minus 1, what about 2c? In 2c you are doing work proportional to the degree of or out degree of v okay so total cost of 2c across all iterations is equal to sum of out degree of v, v belongs to V minus s

$$=\sum_{v\in V-\{s\}}\text{outdegree }(v)$$

Okay you know that the out degree sum is nothing but m right number of edges total number of edges recall from our introduction that the out degree sum is exactly equal to the total number of edges that implies the total cost of 2c across all iterations is order m, therefore the complexity of Dijkstra's algorithm is order n square plus m.

$$O(n^2+m)$$

This is a very simple implementation, very direct implementation. We did not employ any sophisticated data structures. We just used arrays to maintain the information that we want to use and searched in the array to find the minimum v and updated the array by changing the values.

Okay, all these things are done in a very simple manner, this simple implementation has a complexity order n square plus m. There are some interesting data structures that we can employ and improve the complexity of this algorithm, the algorithm is same but instead of maintaining the information of the vertices in V minus S in a simple array instead of maintaining in a simple array. If we use heap or Fibonacci heaps it is possible to improve the complexity significantly and that is what we are going to look at in the discussions related to this in the next session, thank you.