Course: Introduction to Graph Algorithms

Professor: C Pandu Rangan

Department: Computer Science and Engineering

Institute: IISc

Week: 04

Lecture 18 Dijkstra Algorithm P3

Namaskara we have seen the basic ideas behind a Dijkstra's algorithm and a very simple implementation of that with a complexity order n square plus m Dijkstra's algorithm. We use the two arrays d array and P array. And using this when we implemented our idea it ended up in an algorithm with complexity order n square plus m. We are going to use some data structures and improve the complexity. The purpose of a data structure is making the access and processing of the data more efficient. Internally we will organize the information and that organization or the data structure will allow us to carry out the operation more efficiently.

One operation that we are doing repeatedly here is find minimum value among the D vector values and we also delete that one because that vertex is removed from V minus S and then it goes into S and for the next iteration we do not have that vertex at all therefore that value is a deleted delete min okay. We are going to find a min we are going to delete the min these two we are doing on the D vector of values. okay so there is a set of values, the set of values are Dv such that v belongs to V minus S this is a set of values.

$$\{D[v] \mid v \in V - S\}$$

These values we maintained in an array and when we maintain in an array it resulted in certain complexity. We not only do this we also do an updating, the updating involves decreasing of the value, we update for example D of u is equal to D of v plus weight of v u weight of v u.

$$D[u] = D[v] + w(v,u)$$

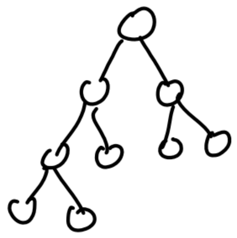
This we are doing if D of u is greater than D of v plus weight of v u.

$$D[u] > D[v] + w(v,u)$$

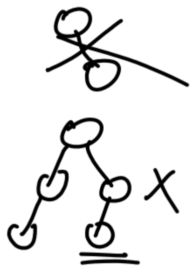
That means every updating involves decreasing of the value this implies every update of a Du value involves in decreasing the value okay. Every update of Du value involves in decreasing the value okay.

In an array decreasing of the value is done simply by rewriting the value. Let us say I have in an array D7 I have 43 if I want to decrease it to 35 I rewrite D7 equal to 35 in one step decreasing of the value is easy to achieve but we are now going to look at a data structure in which all these operations are to be done in a careful manner okay.

One data structure that implements finding the minimum and deleting the minimum in an easy way is heap. Heap is a well known data structure, this is a binary tree and in this binary tree all leaves are placed as much to the left as possible, so for example this is an example of a heap

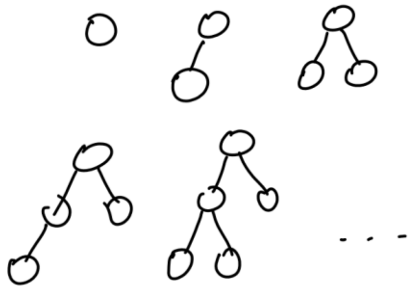


This is not a heap. this is not a heap



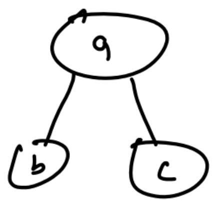
Because this leaf cannot be there it has to be as much to the left as possible in the last level which means the heaps are unique there is only one heap structure for a given value.

A heap of size 1 can look only like this a heap of size 2 can look only like this and a heap of 3 has this structure there is no other way. right and a heap of 4 this level is full and in the next level it goes to the left most so 2, 3, 4, 5 must look like this and so on.



So in this way a heap is built level by level filling a level from left to right every level should be filled from left to right. This heap has got several interesting properties, a heap with n nodes will have theta log n levels And not only that this is a structure of a heap the way in which we are going to store the values, the numbers they are said to satisfy what is known as heap property, heap order property. Since we are interested in obtaining a minimum and deleting a minimum we will have what is known as Min heap order.

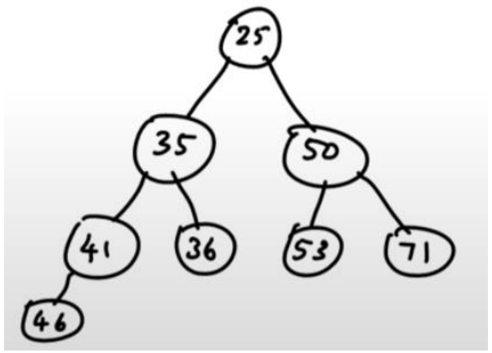
So a value at a node will be smaller than the values at its children a will be less than b and a will be less than c.



$a < b,$

$a < c, .$

b and c can have any relative values b may be smaller than c, c may be smaller than b it does not matter it is only with parent the relation is with parent. Parent must have a smaller value than the children okay. So a must be less than b and a must be less than c this is called the heap order so if you have let us say 25, 35, 50, 41, 36, 53, 71, 46 is an example of a heap. You can see that value at a node is smaller than the values at its children okay.

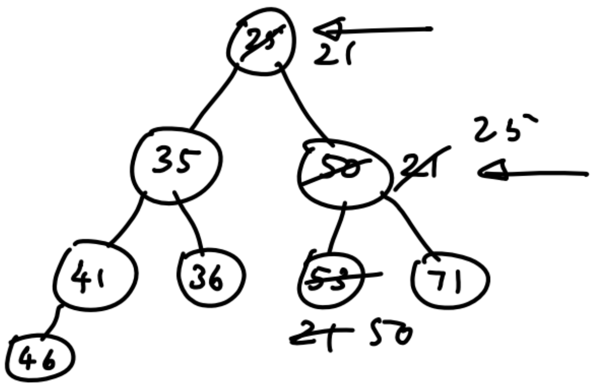


Now minimum will always be at the root the reason is simple right. So smaller values will always be located at higher level smaller values cannot be at a lower level right. because smaller values are at higher level the smallest value is at the highest one namely at the root. This property must be satisfied only then you can do the following operation find Min is very simple just to read the root value that is the minimum then you have to delete This delete Min can be done in order log n time this delete Min can be done in order log n time. Delete Min can be done in time proportional to number of levels since the number of levels is order log n deleting will take log n time.

These are all the standard facts about the heap you can refer any book on data structures to see the procedure for delete Min operation. what is the time it would take for decreasing a value. Suppose you want to decrease one of the values in the process of updating okay suppose I want to change this 53 to 21, I want to decrease it I have already decreased it now but 21 cannot stay there the reason is it would violate the heap order property heap order property is a smaller value must be at the top and if children must have larger values 50 is a larger than 21, 50 cannot sit at a parent of a node that has 21. But modifying this is easy right. All you have to do is bring 50 down and move 21 up.

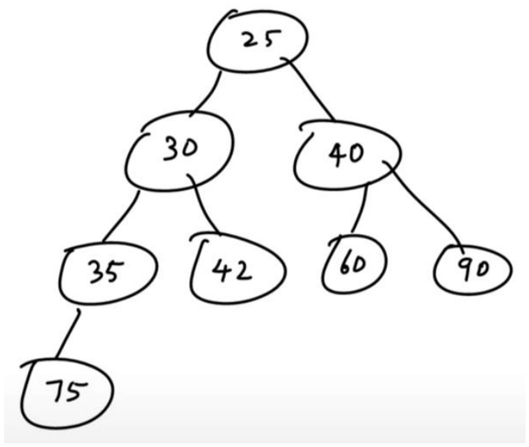
So bring 21 here and now what happens is you can see that 21 is sitting on the top of 50 and 71 that is okay. A smaller value can be in a node and larger values are at its children. Therefore, whenever there is a violation a very simple way to repair the violation is swap with parents exchange with parent. So 21 goes up and 50 comes down in that exchange process. While it has set right the property here it creates a violation with respect to 25.

You cannot have 21 in the bottom and 25 at a level above that or at its parent you cannot have a node containing 21 and its parent having a larger value how to repair that swap. So bring 21 here and 25 here. and now you are at a root, root does not have any parent there would not be any more violations.



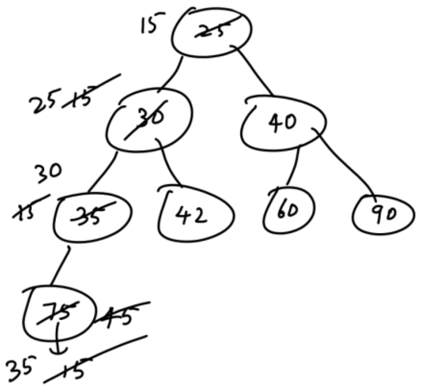
So if the value has reached the root the process would stop. So decreasing of key can result in the following sequence of operations, the node which is having a current value will get an updated value after getting that updated value which is smaller it may move up in the tree, level by level because you are swapping with the parent node and if there is still a problem you would further swap with its parent in this way the value will keep moving up until it can stay there without any violation.

I will give just couple of more examples of, so 25 okay let me write with bigger.



So now you see that this is a heap if I reduce 75 to 45 let us say I am reducing 75 to 45 absolutely no problem I can put 45 here. 45 is still larger than 35 no violation it is a larger number 35 is smaller 45 is larger 45 can stay there. There is no violation, therefore if you are decreasing of the value is something like this after the decreasing of the value. The value that is newly come will stay there okay but suppose you want to decrease 75 to 15, 75 to 15 in that case 75 is to be updated to 15, 15 is smaller than 35 it cannot be there so it has to move up, so 15 comes here and 35 comes over here.

Again 15 cannot sit as a child of 30 that is not possible so swapping 15 is brought here and 30 is brought here again 15 cannot stay as a child for 25, so 25 is brought and 15 so it can go all the way up to the root.



It can stay there it can stop in between or it can go all the way up to the root so it may jump over as many times as number of levels in the tree. The journey upwards is done level by level. So the number of levels being log n, the cost of decrease value of a particular value suppose you want to implement this. If the values are all stored in a heap like this, the decrease value takes order log n time, therefore if I decide to maintain the values what are the values I have to maintain D of v for v belong to V minus S.

If I want to maintain these values in an array it has certain complexity, if I maintain them in a heap okay, finding the Min you can do in one step it is there in the root of the heap. Delete Min because you have to move it off, that is going to take log n this is for delete. In every iteration you are performing one delete Min operation, therefore the total cost for moving to S, every step you move one means you are deleting that will be n log n, it is upper bound by n log n. Now what happens is in step 2c you are updating, that update can happen for every edge, right, each edge each neighbor, may have a larger value therefore you may have to decrease the value and any decreasing of the value you cannot simply decrease the value and leave it there because we are maintaining it in a heap all heap properties must satisfy, which means I have to move the value up notice that I do not have to do any of these things in an array. But here I want the values to be stored in a heap means after decreasing it may take log n time to finally settle. So there may be total cost of updates is m times log n, because this is the number of edges total number of edges each edge may trigger a decrease value operation when you have the decrease value operation it will take log n time therefore the total, therefore what is the total complexity the total complexity is order n log n plus m log n

$$O\big((n+m)\log n\big)$$

How does this compare with our previous array based implementation which is n square plus m

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

How do you compare these two, which one is better well you cannot definitely say which one is better it depends on the graph. If the graph is a sparse graph sparse means less number of edges, for example planar graphs okay have got only order n edges there are many graphs which will have let us say order n edges let us say m is order n this is sparse graph.

$$m = O(n)$$

If m is order n, the array based one is order n square plus n which is still order n square. The n log n plus m log n becomes order n log n because m is order n.

$$m = O(n) \rightarrow \; O(n^2+n) \rightarrow O(n^2)$$

$$O(n\log n + m\log n) \rightarrow O(n\log n)$$

So you see that our array based implementation is n square heap based implementation is n log n so for sparse graph heap based implementation is faster okay so let us for sparse graphs okay heap based implementation is faster. For dense graph, what is a dense graph lot of edges are there for example m equal to order n square take complete graph complete graph has got n times n minus 1, edges all possible edges it is a directed graph.

So, there could be order n square edges if this is the case you can see that order n square plus m algorithm is order n square plus n square which is order n square.

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

Whereas order n log n plus m log n becomes order n log n plus n square log n this is a smaller term this is a dominant term therefore this will be order n square log n lower order terms do not contribute in big O complexity. So this is n square whereas this is n square log n like algorithm that is slower okay.

$$O(n \log n + m \log n) \to O(n \log n + n^2 \log n) \to O(n^2 \log n)$$

So, the conclusion is, for dense graph array based implementation is faster than heap based implementation. So for sparse graph heaps are better for dense graph arrays are better that is the kind of a situation it is a new idea but the implementation is not uniformly better it is definitely more efficient on certain cases but not all cases, however the researchers have come up with a brilliant data structure, it is way too complex and that is called the Fibonacci, in Fibonacci heap m decrease value operations will take only order m times.

The total number of you are doing several decrease key operation for each edge it may trigger a decrease value operation alright. For some it will be more for some it will be less it would not be uniformly log n but total it can be shown, is order m. The other complexity is n log n much like heap, therefore in Fibonacci heap the complexity will be order n log n plus m

$$O(n \log n + m)$$

This is better than both order n square plus m or order n log n plus m log n.

$$O(n^2 + m) \qquad\qquad O(n \log n + m \log n)$$

Definitely this is better than both. It is a complicated and fairly advanced data structure the mathematical analysis of that is also very sophisticated and we are not going to discuss the Fibonacci heap based implementation but just as an information I just want to state that we have a clever data structure which results in a very very efficient implementation of Dijkstra's algorithm.

The complexity will be order n log n plus m we have not discussed that and then we are not going to look into that data structure or those implementation okay, just a fact for your information. So with these remarks I conclude my discussions on Dijkstra's algorithm okay, remember the Dijkstra's algorithm works only if the graphs have positive weight, okay,

$$w(e) \geq 0$$

Dijkstra's algorithm works only if all weights are positive or non-negative okay. The graph can have any structure, but all weights must be positive for this to work. Dijkstra's algorithm is guaranteed to work only on the graphs that have got positive weights alright.

I conclude at this point. Thank you.