Minimal Spanning Tree

Edited slides of Prof Pallab Dasgupta

Pallab Dasgupta
Professor,
Dept. of Computer Sc. & Engg.,
Indian Institute of Technology Kharagpur

Minimal Spanning Tree

- Let G = (V, E) be a weighted graph, where $\omega(e)$ denotes the weight of edge e.
  - The weight of a spanning tree T of G equals the sum of the weights of the N – 1 edges contained in T
  - T is called a *minimal spanning tree* if no spanning tree has a smaller weight than T.

- *If all edge weights are different, there is only one MST*

Two Properties of MST’s

Cycle Property: For any cycle C in a graph, the heaviest edge in C does not appear in the minimum spanning tree

- Used to rule edges out

Cut Property: For any proper non-empty subset X of the vertices, the lightest edge with exactly one endpoint in X belongs to the minimum spanning forest

- Used to rule edges in

Simplifying assumption : All edge costs are distinct.

Let S be any subset of vertices, and let e be the min cost edge with exactly one endpoint in S.

Then the MST T* contains e.

Pf. [by contradiction]!

Suppose e does not belong to T*.

Let's see what happens.

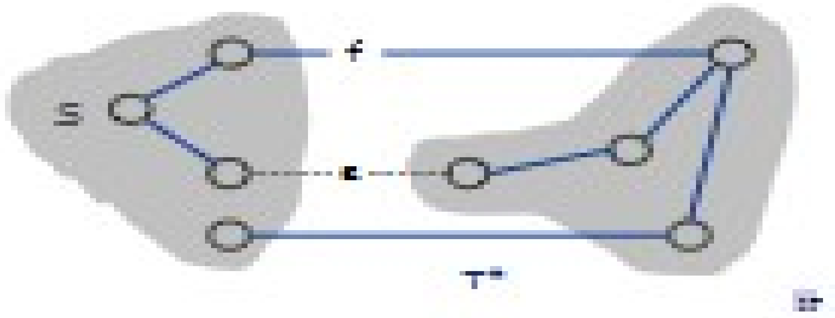
Adding e to T* creates a (unique) cycle C in T* since

Some other edge in T*, say f, has exactly one end point in S.

T = T* + {e} - {f} is also a spanning tree.

Since c(e)<c(f), cost(T) < cost(T*).

This is a contradiction.



The notion of a *fragment*

- A fragment is a subtree of a MST
- If F is a fragment and e is the least-weight outgoing edge of F, then F U {e} is a fragment
- Prim's Algorithm:
  - Start with a single fragment and enlarges it in each step with the lowest-weight outgoing edge of the current fragment
- Kruskal's Algorithm:
  - Starts with a collection of single-node fragments and merges fragments by adding the lowest-weight outgoing edge of some fragment

Gallager-Humblet-Spira Algorithm

- Distributed algorithm based on Kruskal's algorithm.
  Has no separate union find datastructure. However an edge is part of MST if it joins two trees with different fragment names. If both have the same name then the edge is an internal edge causing a cycle.
- <u>Assumptions:</u>
  - Each edge e has a unique edge weight $\omega(e)$
  - All nodes though initially asleep are awaken before they start the execution of the algorithm. When a process is woken up by a message, it first executes the local initialization procedure, then processes the message

Gallager-Humblet-Spira Algorithm

- **Algorithm Outline:**

1) Start with each node as a one-node fragment
2) The nodes in a fragment cooperate to find the lowest-weight outgoing edge of the fragment
3) When the lowest-weight outgoing edge of a fragment is known, the fragment will be combined with another fragment by adding the outgoing edge, in cooperation with the other fragment
4) The algorithm terminates when only one fragment remains

Gallager-Humblet-Spira Algorithm

- **<u>Notations and Definitions:</u>**

  1) ***Fragment name.*** **To determine whether an edge is an outgoing edge, we need to give each fragment a name.** // similar to having the same root node in union find for all nodes in the same fragment

  2) ***Fragment levels.*** **Each fragment is assigned a *level*, which is initially 0 for an initial one-node fragment.**

  3) ***Combining large and small level fragments.*** **The smaller level fragment combines into the larger level fragment by adopting the fragment name and level of the larger level fragment. Fragments of the same level combine to form a fragment of a level which is one higher than the two fragments. The new name is the weight of the combining edge, which is called the *core edge* of the new fragment.**

Gallager-Humblet-Spira Algorithm

- <u>Summary of combining strategy:</u> A fragment F with name FN and level L is denoted as F = (FN, L); let e_F denote the lowest-weight outgoing edge of F.
  - **Rule A.** If e_F leads to a fragment F′ = (FN′, L′) with L < L′, F combined into F′, after which the new fragment has name FN′ and level L′. These new values are sent to all processes in F
  - **Rule B.** If e_F leads to a fragment F′ = (FN′, L′) with L = L′ and $e_{F'} = e_F$, the two fragments combine into a new fragment with level L+1 and name $\omega(e_F)$. These new values are sent to all processes in F and F′.
  - **Rule C.** In all other cases fragment F must wait until rule A or B applies. So if joining with a lower fragment then the larger fragment will wait.

Gallager-Humblet-Spira Algorithm

- **<u>Node and link status:</u>**
  - **$status_p[q]$: Node p maintains the status of the edge pq.**

    Initially an edge is basic. The edge is branch if in the fragment and *reject* if the edge is known not to be in the MST.
  - **$father_p$: For each process *p* in the fragment, *$father_p$* is the edge leading to the core edge of the fragment.**
  - **$state_p$: State of node *p* is *find* if *p* is currently engaged in the fragment's search for the lowest-weight outgoing edge and *found* otherwise. Initially it is in state *sleep*.**

var $state_p$: (*sleep, find, found*) ;

$statch_p[q]$: (*basic, branch, reject*) for each $q \in Neigh_p$;

$name_p$ *//name of the fragment*, $bestwt_p$ *//best local wt p knows about* : real ;

$level_p$: integer ;

$testch_p$ *//testchannel: to test if edge belongs to same fragment*, $bestch_p$ *//best outgoing edge from fragment rooted at p*,

$father_p$ *//pointer pointing to core edge* : $Neigh_p$;

rec_p *//number of msgsreceivedwhile* : integer;

Initialisation code. Process does this first on waking up

(1) As the first action of each process, the algorithm must be initialized:

begin let *pq* be the channel of *p* with smallest weight ;

$status_p[q]$:= *branch* ; $level_p$:= 0 ;

$state_p$:= *found* ; rec_p := 0 ;

send ⟨connect, 0⟩ to *q* *//msg is of form <connect, level>*

end

GHS Algorithm: Part-1

(2) Upon receipt of ⟨connect, L⟩ from *q*:

begin if $L < level_p$ then (* Combine with rule A *)

begin $status_p[q]$:= *branch* ;

send ⟨initiate, $level_p$, $name_p$, $state_p$⟩ to *q* *//every node in subtree of q needs to update level, name and state*

end

else //L=levelp as connect not called on L>levelp due to test code

if $status_p[q]$ = *basic* *//other side did not find the same min outgoing edge*

then (* Rule C *) process the message later

else (* Rule B *) send ⟨initiate, $level_p + 1$, $\omega(pq)$, *find*⟩ to *q*

end

Question ---

else //L>=levelp

if $status_p[q] = basic$

then (* Rule C *) process the message later

else (* Rule B *) send ⟨initiate, $level_p + 1$, $\omega(pq)$, *find*⟩ to q

end

Assume for now that the future code ensures that statusp[q]=basic only if p's level is smaller (this is taken care of by test function). Hence you can assume now Rule B applies in the else part.
How does the code make sure that the two fragments of equal size are connected only if both sides have identified the same min outgoing edge?

GHS Algorithm: Part-1

(3) Upon receipt of ‹initiate, L, F, S〉 from *q*:

begin $level_p := L$; $name_p := F$; $state_p := S$; $father_p := q$;

$bestch_p := udef$; $bestwt_p := \infty$;

forall r $\in$ $Neigh_p$: $statch_p[r] = branch \wedge r \neq q$ do

send ‹initiate, *L, F, S*〉 to *r* ;

if $state_p = find$ then begin $rec_p := 0$; *test* end

end

//Procedure test is called on a node asking it to check for its min outgoing edge followed by reporting to its parent the min outgoing edge among those reported by its subtree compared to its own.

Testing the edges - To find its lowest-weight outgoing edge

- To find its lowest-weight outgoing edge, node *p* inspects its outgoing edges in increasing order of weight.
- To inspect edge *pq*, *p* sends a ‹test, $level_p$, $name_p$〉 message to *q* and waits for an answer
  - A ‹*reject*〉 message is sent by process *q* if *q* finds that *p*'s fragment name is the same as *q*'s fragment name. On receiving the ‹*reject*〉 message, *p* continues its local search.
  - If the fragment name differs *q* sends an ‹*accept*〉 message provided the level is smaller (read below)
  - The processing of a ‹*test, L, F*〉 message is deferred by *q* if $L > level_q$ because *p* and *q* may actually belong to the same fragment, but the ‹*initiate, L, F, S*〉 message has not yet reached *q*.

A simple optimization

- To inspect edge *pq*, *p* sends a ‹test, $level_p$, $name_p$〉 message to *q* and waits for an answer
  - A ‹*reject*〉 message is sent by process *q* if *q* finds that *p*'s fragment name is the same as *q*'s fragment name.
  - If the edge *pq* was just used by *q* to send a ‹*test, L, F*〉 message then *p* will know (in a symmetrical way) that the edge *pq* is to be rejected. In this case, the ‹*reject*〉 message need not be sent by *q.*

=> Q sent test, P sent test, P sent reject , Q need not send reject as P knows.

Not FIFO channel so even possible that Q sent test, P sent test, P sent reject but reject received before test from P so by the time P's test gets to Q, Q has already marked qp as reject.

Similarly for Accept? No..since then accept gets to the root and starts a connect with a smallernode

Test node – check who is your min outgoing edge

(4) **procedure** *test*

begin if $\exists q \in Neigh_p$: $statch_p[q] = basic$ **then**

begin let $testch_p := q$ **when**

$statch_p[q] = basic$ **and** $\omega(pq)$ **minimal ;**

send $\langle$test, $level_p$, $name_p\rangle$ **to** $testch_p$

end

else begin $testch_p := udef$; *repo*

end

Basic edge can be to own fragment
Or to other fragment. If already branch or reject then we cant consider it for the min outgoing edge

Test procedure is when a node is asked to check its incident basic edges in increasing order of weight to find an edge going out to another fragment.

Report: Checks if it has got <report,w> from all its children and has completed local computation in which case it sends the min wt edge to its parent.

GHS Algorithm: Part-2

(5) **Upon receipt of** ⟨test, *L*, *F*⟩ **from** *q*:

begin if $L > level_p$ then (* Answer must wait *)

process the message later

else if $F = name_p$ then (* internal edge *)

begin if $statch_p[q] = basic$ then $statch_p[q] := reject$;

if $q \neq testch_p$

then send ⟨reject⟩ to *q*

else *test*

end

else send ⟨accept⟩ to *q*

end

As not FIFO, consider the sequence p sent test, q sent test, q rejects p But reject reaches before its test. So when q's test arrives, pq already marked as reject

Optimisation: if p has sent q a test already meanwhile, then don't have to send reject to q as q will know

If q=testchp it means p had sent a test which got rejected. node p has to resume its test.

Inconsistent test queries - question

Suppose we have two equal sized fragments F1 and F2 which have sent each other the initiate msg with status find. The initiate msg as we know sends initiate and test msgs down the tree to further children. Meanwhile, suppose that an initialised node of one fragment F1 sends a test to an uninitialised node of F2. F2 will find its fragment name to be different from the requested node and will accept. Issue?

Similar case : Suppose we have two unequal sized fragments F1 (larger) and F2 where F1 has sent F2 an initiate msg with status find. The initiate msg as we know sends initiate and test msgs down the tree to further children. Meanwhile, suppose a node of F1 sends a test to an uninitialised node of F2. F2 will find its fragment name to be different from the requested node and will accept. Issue?

GHS Algorithm: Part-2

(6) **Upon receipt of** ⟨accept⟩ **from** *q*://local search of outgoing edge is done.

begin $testch_p$:= *udef* ; //indicate that local search is over

if $\omega(pq) < bestwt_{p//bestp}$ *is the best wt p has seen among its own and its subtree*

then begin $bestwt_p := \omega(pq)$;

$bestch_p := q$

end ;

report

end

Each time a <report,w> msg reaches, the value of bestwtp is updated. The current bestwtp is the best among what it has received so far. Each 'report' checks if all msgs <report,w> msgs have reached and local computation is complete in order to send result to parent.

(7) **Upon receipt of** ⟨reject⟩ **from** *q*:

begin if $statch_p[q] = basic$ then $statch_p[q] := reject$;

test

end

Reporting the lowest-weight outgoing edge

- The lowest-weight outgoing edge is reported for each sub tree using ‹report, ω⟩ messages
  - Node p counts the number of ‹report, ω⟩ messages it receives, using the variable rec_p.
- The ‹report, ω⟩ messages are sent in the direction of the core edge by each process.
  - The messages of the two core nodes cross on the edge; both receive the message from their father
  - When this happens, the algorithm terminates if no outgoing edge was reported. Otherwise a ‹connect, L⟩ message must be sent through this edge.

Sending the connect

- If an outgoing edge was reported, the lowest-weight outgoing edge is found by following the *bestch* pointer in each node, starting from the core node on whose side the best edge was reported
- This is done by sending a ⟨changeroot⟩ message.
- When the ⟨changeroot⟩ message arrives at the node incident to the lowest-weight outgoing edge, this node sends a ⟨connect, L⟩ message via the lowest-weight outgoing edge.

GHS Algorithm: Part-3

procedure ***report***: //if the number of msgs received is equal to its children and its own search has concluded(testchp-udef) then put state as found and send report to father. A report call is a check to see if all <report,w> msgs have reached.

begin if $rec_p = \#\{\, q : statch_p[q] = branch \wedge q \neq father_p \,\}$

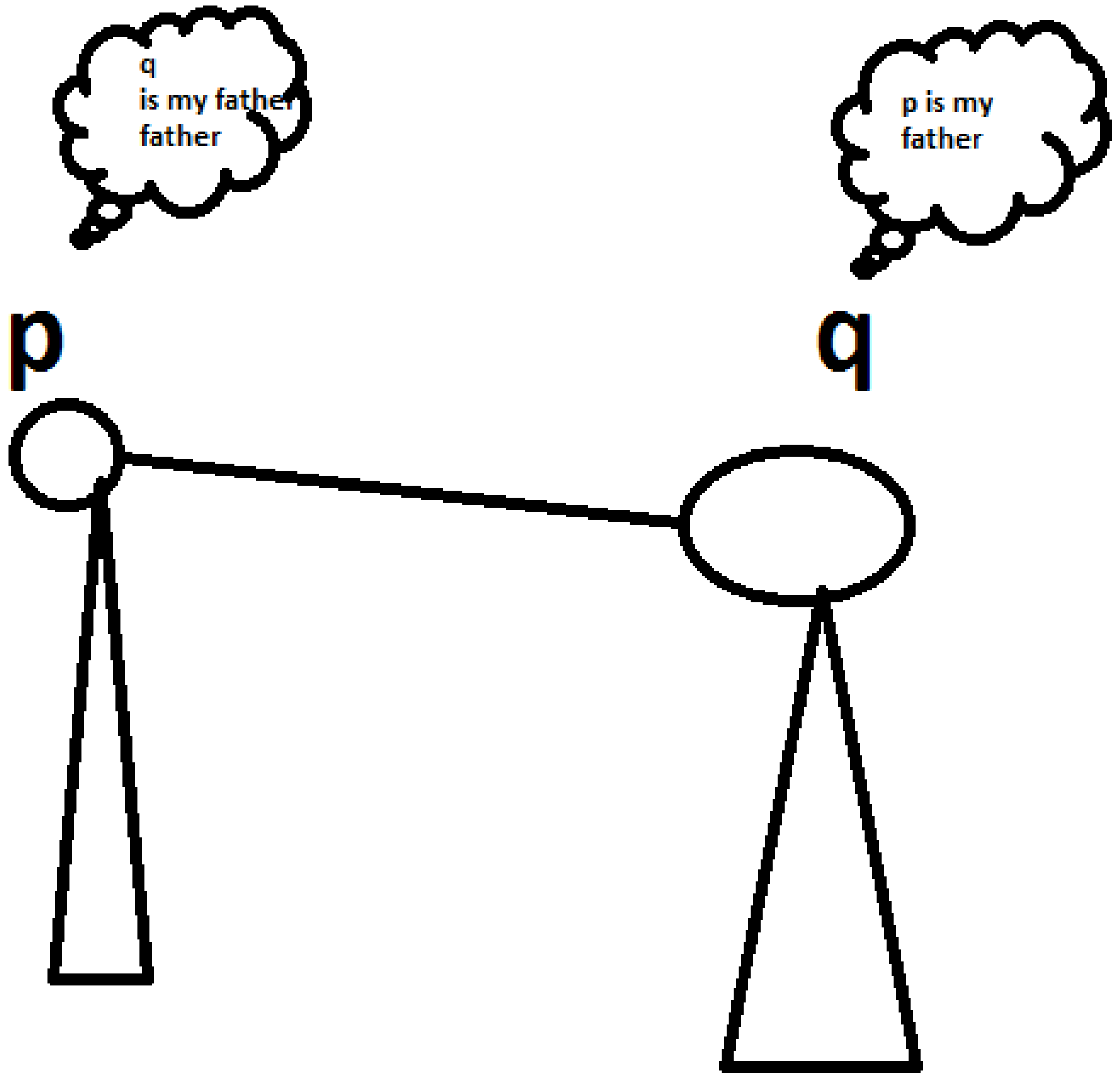
and $testch_p = udef$ **then**

begin $state_p := found$;

send $\langle$report, $bestwt_p\rangle$ **to** $father_p$

end

end


q
is my father
father
p is my
father
p
q

GHS Algorithm: Part-3

(9) Upon receipt of ‹report, ω› from q:

begin if $q \neq father_p$ // at the core edge p and q are fathe of eachother. Could have put condn as if pq not core edge

then (* reply for initiate message *)

begin if $\omega < bestwt_p$ then

begin $bestwt_p := \omega$; $bestch_p := q$ end ;

$rec_p := rec_p + 1$; *report* // report called again to chk if it has got all msgs to send to parent

end

else (* *pq* is the core edge *)

if $state_p = find$// p is still looking for min outgoing while q's result arrives so make q wait

then process this message later

else //state is found

if $\omega > bestwt_p$//if w<bestwtp then q will do changeroot when it gets <report,w>. Wt w!=bestwtp as unique edge weights

then *changeroot*

else if $\omega = bestwt_p = \infty$ then stop //when both don't have anything to report then both report infinity implying single fragment. Then stop

end

GHS Algorithm: Part-3

(10) procedure *changeroot*://if you are the node having the min edge then you send connect. Else you send changeroot to your child trhough which you got the min edge

begin if $statch_p[bestch_p]$ = *branch*

then send ‹changeroot〉 to $bestch_p$

else begin

$statch_p[bestch_p]$:= *branch* *//this takes care of p and q connecting on the same edge if both have same levels by rule2*

send ‹connect, $level_p$〉 to $bestch_p$;

end

end

(11) Upon receipt of ‹changeroot〉 :

begin *changeroot* end

Complexity

The Gallager-Humblet-Spira algorithm computes the minimal spanning tree using at most 5N logN + 2|E| messages

- Each edge is rejected at most once and this requires two messages (test and reject). This accounts for at most 2|E| messages.

- At any level, a node receives at most one initiate and one accept message, and sends at most one report, one changeroot *or* connect message, and one test message not leading to a rejection. For logN levels, this accounts for a total of 5N logN messages.