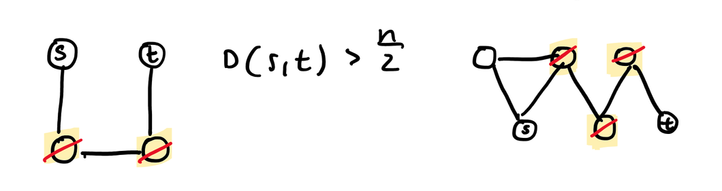
CS 180 HW 3

Friday, October 23, 2020 8:53 PM

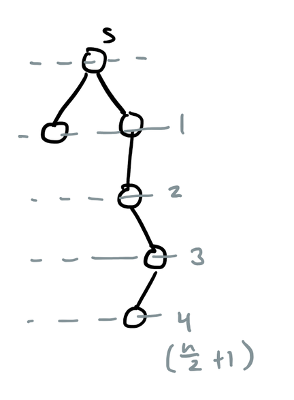
Amy Seo

Dis. 1D
10 - 11:50 AM
Song

1. $G=(V,E)$, n nodes, m edges



*Proof by contradiction: Suppose that an n-node undirected graph $G=(V,E)$ contains 2 nodes s and t such that the distance between s and t is strictly greater than $\frac{n}{2}$, and that there does not exist a node that when deleted, will destroy all s-t paths.



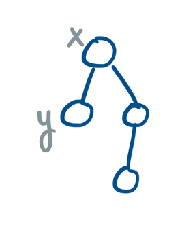
- Since the distance between s and t is greater than $\frac{n}{2}$, t must be at level $n/2 + 1$ at the least in the BFS tree.
- In levels 1 to $\frac{n}{2}$, there are a total of at most $n-2$ nodes, as we exclude s and t
- It follows that if each of these levels from 1 to $\frac{n}{2}$ has more than one node, there are $2(\frac{n}{2})+2$ nodes in the graph, which is a contradiction as we only have n nodes.
- Thus, there must be at least one level from level 1 to $\frac{n}{2}$ where there is only one node.
- Then, deleting this one node would result in no path from s to t.

∴ By contradiction, there must be a node v such that deleting it would destroy all paths between s and t.

Algorithm:
- perform a BFS starting at s
  - at each level, check how many nodes there are
  - when we come across a level between 1 and $\frac{n}{2}$ with only one node, output this node.
    - ↳ from our proof*, it's guaranteed to have a level between 1 and $\frac{n}{2}$ w/ only one node

Time complexity: $O(m+n)$ because this is a BFS, so we will visit at most all n nodes and traverse all m edges.

2. Proof by contradiction: Suppose that BFS and DFS of a connected graph G rooted at a node u produces the same tree T such that T is missing an edge in G.

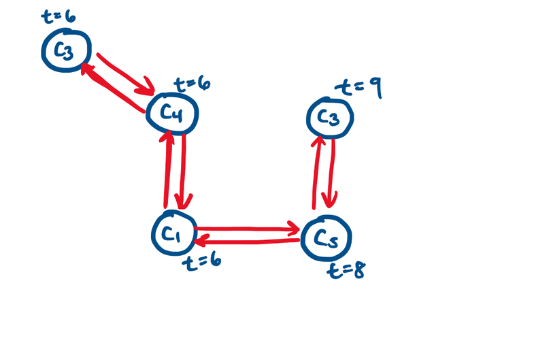


- Then there must be at least one edge $e=(x,y)$ that is in G, but not in T.
- Since T is a BFS / DFS tree, it is guaranteed that we will visit every node in G by the nature of these searching algorithm.
- It follows then, that we will visit either x or y in DFS first, and the other node would be the child of it.
  - ↳ Then we must visit edge $e=(x,y)$
- It also follows that when we visit x in BFS (w/out loss of generality), y must be in the next level.
- Thus, w/out loss of generality, x must be the parent of y and they only differ by one level.
- This means that $e=(x,y)$ is in T, which is a contradiction.

∴ G must equal T.

3. Example:

Inputs C_A, $t=x$
- infection: $(C_4, t=5)$
- trace data: [$(C_1, C_2, t=2)$,
  $(C_2, C_3, t=3)$,
  $(C_3, C_4, t=6)$, ←
  $(C_4, C_1, t=6)$,
  $(C_1, C_5, t=8)$,
  $(C_5, C_3, t=9)$
  ]



- check: $(C_2, t=9)$ → FALSE b/c C_2 is not in path of C_3 once it's infected
  C_b, $t=y$

Algorithm: Suppose we are given a computer C_A that will be infected at time x and a computer C_B that we need to check is infected or not at time y, along with an array of trace data.

- If $y < x$ return false (C_B cannot be infected before C_A)
- Initialize an empty set N to hold tuples (C_o, t_o) which is a computer C_o at time t_o with its own set of directed edges.
- Initialize an empty set R to hold the most recent interactions of a computer
- For each triple (C_p, C_q, t) in the trace data in order where $t \geq x$ and $t \leq y$:
  - Add (C_p, t) and (C_q, t) to N
  - Add (C_p, C_q) as an edge to (C_p, t)
  - Add (C_q, C_p) as an edge to (C_q, t)
  - For each C_p, C_q (denote current one as C_r)
    - if C_r is in R as (C_r, t_r)
      - ↳ add directed edge from (C_r, t_r) to (C_r, t)
      - ↳ update it to be (C_r, t)
    - else
      - ↳ add (C_r, t) to R
      - ↳ if C_r is C_A
        - ↳ set C_r to be the start node
- BFS starting at our start node C_A
  - if we ever come across C_B
    - ↳ return True
- return False

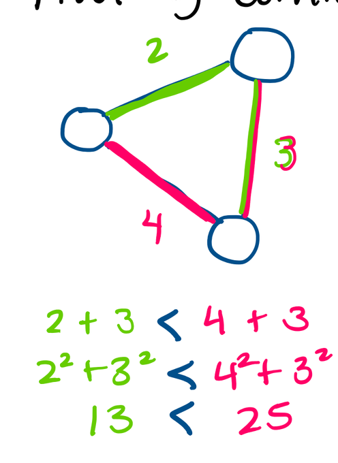
Proof by contradiction:
- Case 1: Suppose C_B is infected at time y, but our algorithm returns False
  - Since our algorithm returns false:
  - After constructing our graph N that keeps track of all communications in $[x,y]$, and conducting BFS from C_A, C_B was nowhere to be found in its path, which means that C_B could not have been infected at time y.
  - This is a contradiction as C_B must have a path from C_A if it is infected.
- Case 2: Suppose C_B is not infected at time y, but our algorithm returns True
  - Since our algorithm returns true, there must be a path from C_A to C_B in time interval $[x,y]$, which means there are a series of communications from C_A to C_B that will cause C_B to get infected.
  - This is a contradiction as C_B is not infected at time y.

∴ By contradiction, our algorithm always returns whether C_B is infected or not at time

Time complexity: $O(m+n)$ because we will traverse at most all n computers and all m connections.

4. a) claim: True; T will always be the minimum spanning tree.

Proof by contradiction: Let G be a graph where all edges are positive and distinct.



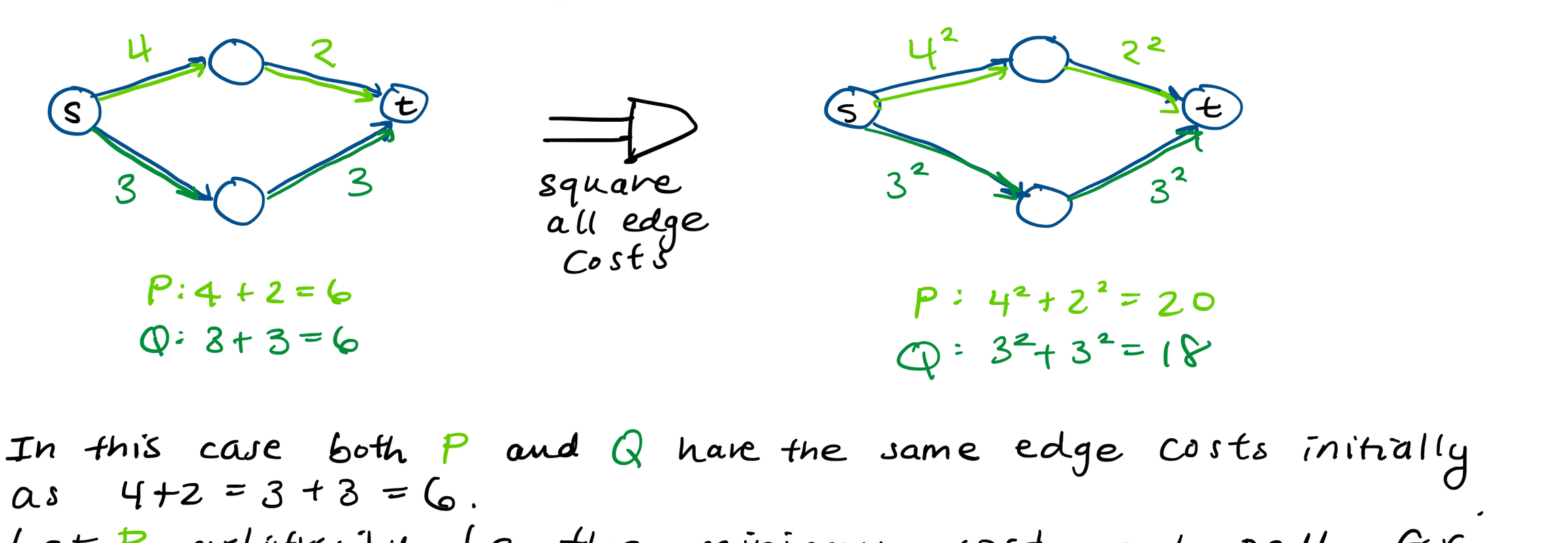
$2+3 < 4+3$
$2^2+3^2 < 4^2+3^2$
$13 < 25$

- Let a, b be the edge costs in T and c, d be the edge costs in T' such that T is the original minimum spanning tree, but T' is now the minimum spanning tree after all edge costs are squared.
- It follows that $(a+b)<(c+d)$ and $(a^2+b^2)>(c^2+d^2)$
- This is a contradiction because if $(a+b)<(c+d)$, a^2+b^2 must be less than c^2+d^2.
- Thus, T will remain as the minimum spanning tree even after all edge costs are squared.

∴ By contradiction, T will always remain the minimum spanning tree

b) Claim: False; P will not always be a minimum-cost s-t path for this new instance.

Counterexample: Let G be a directed graph



P: $4+2=6$
Q: $3+3=6$

P: $4^2+2^2=20$
Q: $3^2+3^2=18$

- In this case both P and Q have the same edge costs initially as $4+2=3+3=6$.
- Let P arbitrarily be the minimum-cost s-t path for this first instance.
- when the edges are squared, $4^2+3^2 > 3^2+3^2$.
- Thus, Q is the minimum-cost s-t path now and not P.

∴ By counterexample above, P will not always be the minimum cost s-t path.

5. Suppose there are d candidates and an array votes of size v where each vote is an integer from 1 to d.

Algorithm: (assuming $d \neq 0$ and length of votes $\neq 0$)
- Initialize count to 0 and candidate to be the first vote in votes
- For every vote in votes
  - if count is 0
    - ↳ set candidate equal to the current vote
  - if current vote is not equal to candidate
    - ↳ decrement count
  - else
    - ↳ increment count
- Iterate through votes and count how many occurrences of the candidate there are
- if count of candidate is > length of votes / 2
  return candidate
- else,
  there is no winner

Proof by contradiction:
- Case 1: Suppose the actual majority winner is M but our algorithm returned N.
  - Then there must be more occurrences of M compared to N that makes M the majority. Let's denote this as $M_{count} > N_{count}$
  - It follows then that when candidate is set to N in the first loop of our algorithm, there must be a point where it's count reaches 0 and the candidate changes since there are more than N_{count} other elements in the votes array, given that $M_{count} > N_{count}$.
  - Then, N could never be the candidate again as the count of all elements other other than $N > N_{count}$
  - This is a contradiction then because our algorithm could not have returned N.
- Case 2: Suppose there is no actual majority winner but our algorithm returns N.
  - After finding the majority element in the first loop (if it exists), we run another loop to check if the number of occurrences of this candidate is greater than the length of the votes array divided by 2.
  - If it is greater, then we return the candidate N.
  - Then N must occur in the votes array more than (length of votes / 2) times, making N the majority, but this is a contradiction as there is no winner.
- Case 3: Suppose there is a majority winner M but our algorithm returned that there is no winner.
  - By definition, M has to occur in votes more than votes length / 2 times for it to be the majority winner. $M_{count} > v/2$
  - It follows that the first loop must set M to be the candidate by the end of the loop because it's count will be ≥ 1 at the end by the nature of the algorithm since all other candidates in votes amount to less than M_{count} occurrences.
  - when we iterate through to count the number of occurrences of M in the array, we found that M_{count} < (length of votes / 2) but this is a contradiction as M is the majority winner.

∴ By contradiction, this algorithm always returns the majority winner, if there is any.

Time Complexity: We iterate through the list of votes twice, making it $O(2n) = O(n)$.

Space complexity: Since we are making use of 2 variables: candidate and count no matter the value of d or v, our storage is constant so the space complexity is $O(1)$.

6. Assuming unweighted graphs:

a) Let G be a DG.

Algorithm: For each node N in G, perform a DFS starting at N
- instead of keeping track of what nodes are visited, keep track of what edges are visited.
- for each edge, perform DFS recursively to try all possible edges
- Keep track of the distance for each traversal and return the longest one.

Proof by contradiction: Suppose the longest path has a distance of P' but our algorithm returns a path of length P.
- This is a contradiction because our algorithm is brute force and will visit every possible path that exists in G, so P must be the longest distance by the nature of our algorithm.

Time Complexity: $O(ne!)$ because in the worst case, for each node we'd visit e possible edges, then $(e-1)$ possible edges and so on until we try every possible path so the runtime is not polynomial.

b) Let G be a DAG. This is a modified algorithm of topological sort

Algorithm: Let d be an array of (node, distance) tuples that is initially empty.
- While there is a source S
  - For each child C of S
    - if C is in d
      - ↳ set C's distance = max (distance, S's distance + 1)
    - else
      - ↳ add (C, S's distance + 1) to d
  - delete S and all of its outgoing edges
- return the maximum distance in d

ABCD

Proof by Contradiction: Suppose the algorithm returns a distance p such that there is some longer path in G with distance p'
- There must be some node N where there is a longer path to N than the one our algorithm calculated.
- Case 1: N has more than one parent node
  - Then our algorithm would've set N's distance equal to the max of it's current distance and its parent node's distance + 1 for each parent node it has.
  - This is a contradiction as there is a longer path than this maximum distance, which is not possible since all nodes must be visited by its parent as this is a topological sort.
- Case 2: N has only one parent node
  - Then we add (N, parent node's distance + 1) to d.
  - There is no other possible path to N so there is no longer path available.
  - This is a contradiction as there is some longer path than this one we calculated.

∴ By contradiction, our algorithm always returns the longest path in a DAG.

Time Complexity: Since we are doing a topological sort, we are visiting each node and each edge, resulting in a time complexity of $O(m+n)$ where m is the number of edges and n is the number of nodes.