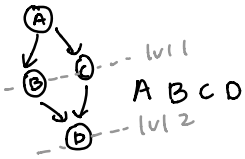
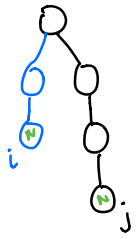
Seo, Amy
505 328 863

1. BFS in a DAG ⇒ topological sort

Algorithm:
- find all sources and add to set S
- while there is a source (while S is not empty)
  - choose a source arbitrarily
    - add this source to the output / visited array
    - delete this node and all of it's outgoing edges
      - if there are any nodes that are now a source (no incoming edges), add them to S



Proof: Suppose our algorithm performs BFS wrong such that it visits a node N on level j when the node is actually on level i such that $j > i$.

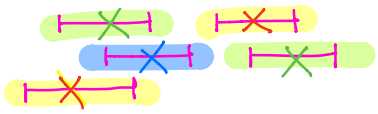


- Then by the nature of our algorithm, we must have found N on level i when performing this topological sort as once all of N's parents are removed, it would be visited. We would not visit N on a further level by nature.
  - Thus, by contradiction, our algorithm would have visited node N on level i.

- Because the graph is a DAG, there will always be source nodes to visit until all nodes are visited.
  - otherwise, the graph is not a DAG.
- thus, it follows that all nodes and all edges will be traversed as it is acyclic.

Time complexity: $O(v+e)$ where there are v nodes and e edges because finding all sources requires us to check all v nodes in the worst case, then we will traverse all v nodes and e edges in the worst case.

2. Assuming the tasks are sorted by start time:



} greedy by start time

Algorithm:
- initialize count to 0
- while there are tasks left:
  - choose the task by earliest start time
  - continue choosing the next earliest task that starts after current task ends until there are no more non-conflicting tasks that can be visited.
  - delete all these visited tasks
  - increment count
- return count

Time Complexity: we will visit all tasks because we are performing this algorithm continuously until all tasks are deleted/visited, so the time complexity is $O(n)$ for n given tasks.

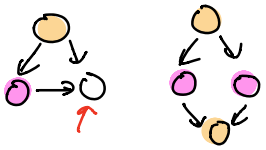
Proof: Suppose our algorithm returns x number of processors when the minimum number of processors N is less than X.

- Then, there must be some processor(s) P in our solution whose tasks can be added on to an existing processor.
- Comparing this to the optimal solution:

- since O' and P' start w/ the same task, there must be a point i where P' terminates and O' continues. Then, we can safely add the remaining tasks of processor O' to P' without any conflicts, removing the need for P.
- But our algorithm would have picked up these extra tasks as they are non-conflicting → contradiction.

∴ Thus, by the greedy nature of our algorithm, we will always return the minimum number of processors.

3. observations



- graph is 2 colorable if it has no odd cycle
- not 2 colorable if it has odd cycle

Algorithm: a modified BFS
- initialize an empty queue Q
- add the start node to Q
- while Q is not empty:
  - for current length L of Q:
    - pop off first node N from Q
    - mark N as visited
    - add all of N's unvisited neighbors to Q
    - if any of these neighbors have an edge between them, there is an odd cycle
      → immediately return that there is no 2-coloring
    - otherwise, color current node N to current color
  - set current color to the other color

Proof: Suppose our algorithm returns the incorrect 2-color result
- Case 1: our graph has no odd cycle but returned an invalid coloring

  - By definition, there must be 2 adjacent nodes of same color that makes it invalid
  - they must be one level apart
  - our algorithm colors nodes by level, so it is a contradiction that nodes in 2 consecutive levels are exactly the same in a graph w/out odd cycles.
- Case 2: graph has odd cycle but returned a 2-color
  - By definition of odd cycle, this graph must have a level where at least 2 nodes have an edge in between them
  - This is a contradiction as our algorithm would have found it during BFS and returned that there is no 2-coloring.

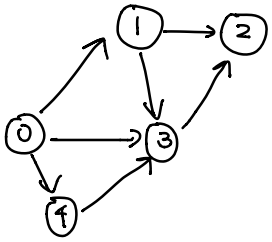
∴ By contradiction, our algorithm always returns a valid 2-color if there is one.

Time complexity: BFS is O(V+E) because we visit each node & edge at most once.

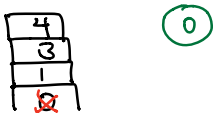
Coloring is O(V) because we color each node at most once.

So total time complexity is O(V+E).

4.

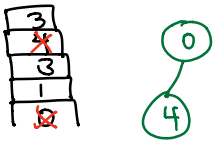


Step 1: vertex 0



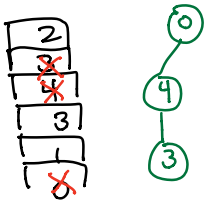
visited = {0}

Step 2: vertex 4



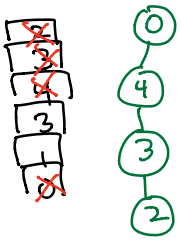
visited = {0, 4}

Step 3: vertex 3



visited = {0, 4, 3}

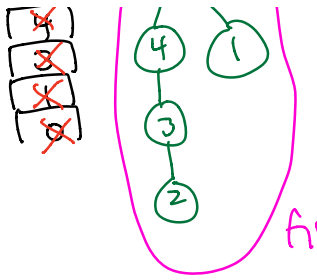
Step 4: vertex 2



visited = {0, 4, 3, 2}

Step 5: vertex 3 already visited (skip)

Step 6: vertex 1



visited = {0, 4, 3, 2, 1}

final DFS tree

5. L = (x1, x2, x3, ...)

- observations: similar to famous problem ? → problem reduction
  - assuming list is unsorted

Algorithm:

- Initialize empty lists: possibleMax and possibleMin
- For every integer in the list L
  - arbitrarily take 2 integers A, B from L
    - compare 2 numbers
      - add larger number in possibleMax
      - add smaller number in possibleMin

  } ☆

- Within possibleMax and possibleMin, keep doing the same comparisons as ☆, but remove the bigger elements in possibleMin and the smaller elements in possibleMax until there is only one number left in each list
- The last number in possibleMax is the maximum
- The last number in possibleMin is the minimum

Proof: Suppose our algorithm returns the wrong answer.

- Case 1: our algorithm returns x for minimum when there exists a number N < x in L, such that N is the minimum in L.
  - Then, there must have been a comparison where N was not less than some other arbitrary number in L, as it does not make it to the final possibleMin list.
  - Then, there must be some other number N' that is less than N that was chosen over N.
  - This is a contradiction as N is the minimum in L.

- Case 2: our algorithm returns y for maximum when there exists a number M > y in L, such that M is the maximum in L.
  - Then, there must have been a comparison where M was not greater than some arbitrary number in L, as it does not make it to the

Ahad possibleMax list.
- Then, there must be some other number M' that is greater than M that was chosen over M.
- This is a contradiction as M is the maximum in L.

∴ By contradiction, our algorithm always finds the minimum & maximum in L.

Time Complexity: It takes $\frac{n}{2}$ comparisons to generate first iteration of possibleMin & possibleMax. Then, we keep comparing half of the list and reducing the problem size:

$$2\left(\frac{n}{2}\right)+2\left(\frac{n}{4}\right)+2\left(\frac{n}{8}\right)+\ldots+1 \approx \frac{3n}{2} \text{ total comparisons.}$$