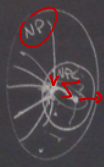
NP Complete (NPC)

1. $L \in NP$
2. L must be NP-HARD
   - Any problem in NP can poly-time reduced to this lang.

$\forall L' \in NP,\ L' \leq_p L$



Some NPC problems can NOT be solved in exp. time.

Fake, b/c we know CircuitSAT can be solved in exp. time and it's NPC/NP HARD

$\Rightarrow$ we can reduce from from any NPC to it in polytime and solve in exp time

reduce

$NP \rightarrow NPC$

reduce

$NPC \rightarrow NPC$

2012

1) 10) A, B, C, $C \leq B$. $f(\cdot)$ is poly-time reduction

$(a \in A \Rightarrow f(a) \in C)$ and $(a \notin A \Rightarrow f(a) \notin B)$

$\Rightarrow a \in A \Rightarrow f(a) \in B$

then $A \leq_p B$

Yes

2012 spring

6) HAMPATH = simple path through every vertex in graph (no cycle)

Degree constrained spanning tree (DST) $\langle G, k \rangle$

Does G have a ST in which no vertex has degree larger than k?

a) Show DST ∈ NP

Certificate = spanning tree T

Verification alg = 1. verify T is a ST $O(V+E)$

2. Check if each vertex has degree $\leq k$ $O(V+E)$

b) Show DST is NP HARD

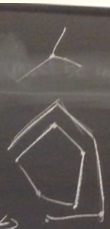
HAMPATH $\leq_p$ DST

$\langle G \rangle$ $\langle G', k \rangle$

i) $G' = G$ $k = 2$

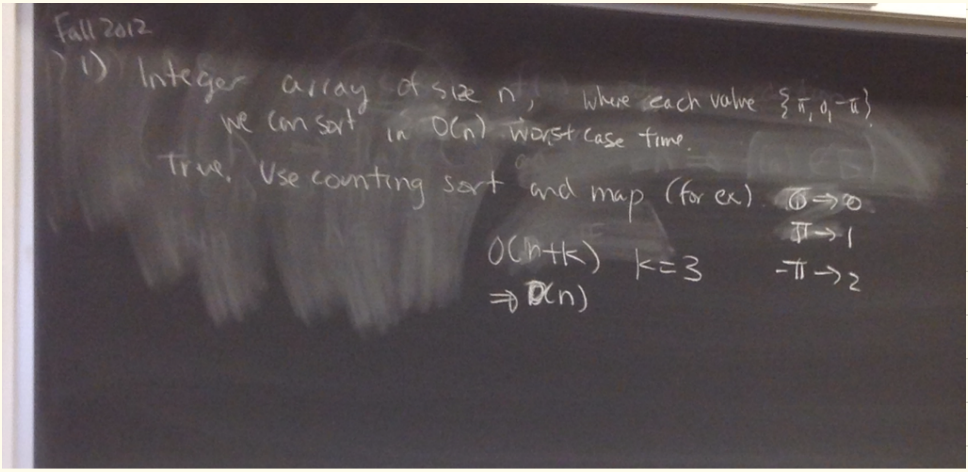
ii) WTS: If HP is "yes" then DST is "yes"

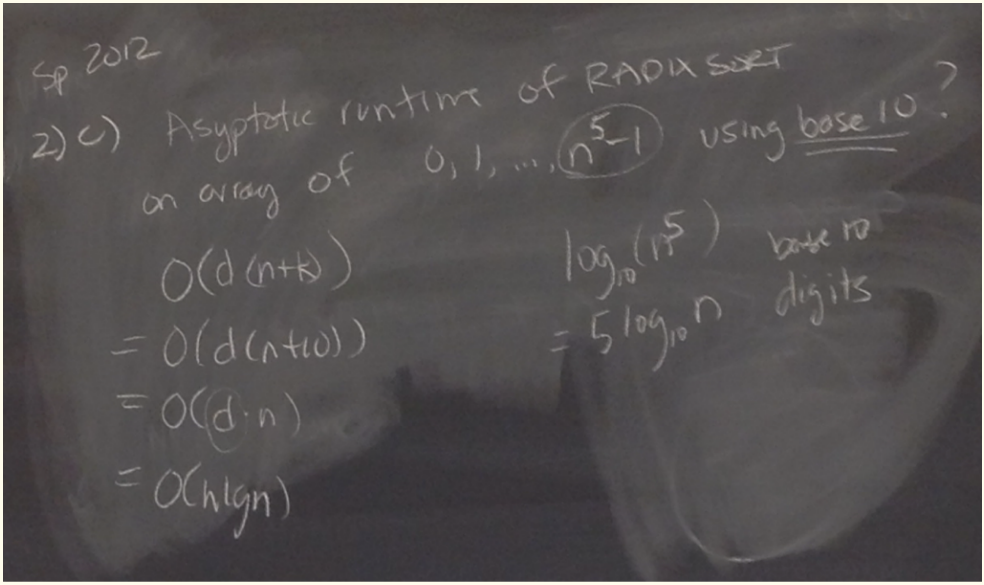
If we have HP then by def'n it goes thru every vertex and each one has deg ≤ 2, therefore it's also a DST $k=2$.

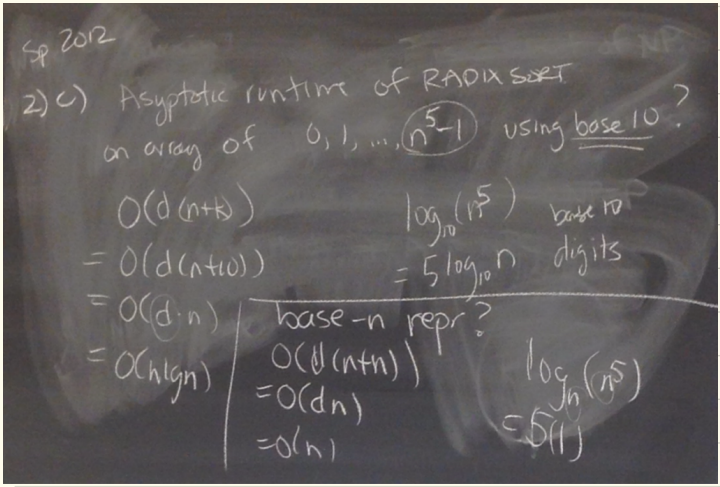


iii) WTS: DST $_{k=2}$ "yes" $\Rightarrow$ HP "yes"

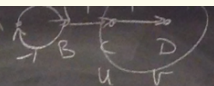
If we have ST w/ deg ≤ 2, then it has be connected and we can't have any branching, therefore it forms a path thru every vertex i.e a hampath.


Fall 2012
1) Integer array of size n, where each value $\{\pi, 0, -\pi\}$.
we can sort in O(n) worst case time.
True. Use counting sort and map (for ex) 0 ⇒ 0
π → 1
-π → 2
O(n+k) k<=3
⇒ O(n)


Sp 2012
2) c) Asyptotic runtime of RADIX SORT
on array of 0, 1, ..., n^5-1 using base 10?
$O(d(n+k))$
$= O(d(n+10))$
$= O(d \cdot n)$
$= O(n \lg n)$
$\log_{10}(n^5)$ base 10 digits
$= 5 \log_{10} n$

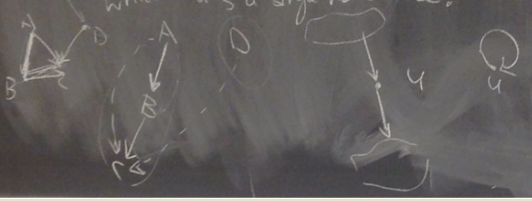

Sp 2012
2) c) Asyptotic runtime of RADIX SORT
on array of 0, 1, ..., n^5-1 using base 10 ?
$O(d(n+k))$
$= O(d(n+10))$
$= O(d \cdot n)$
$= O(n \lg n)$
$\log_{10}(n^5)$ base 10
$= 5 \log_{10} n$ digits
base-n repr?
$O(d(n+n))$
$= O(dn)$
$= O(n)$
$\log_n(n^5)$
$= 5(1)$

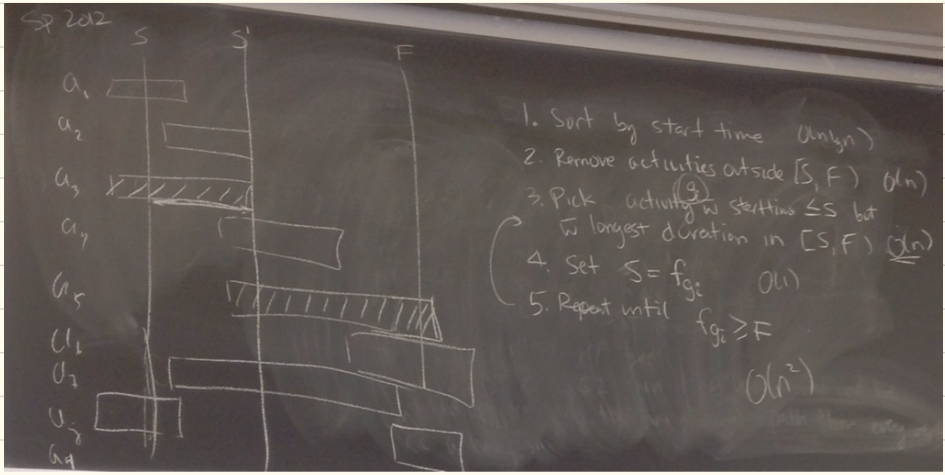
5) If G has neg. weight cycle, there is no pair in G s.t. (u,v) has finite cost



6) Let u be a vertex in G w incoming/outgoing edge. Is it possible for DFS on G to produce DF forest in which u is a single node tree?

SSSP





a) Subproblem.

Steps 2-4, reduce problem (remove activities, reduce interval)

b) Greedy is part of an optimal sol'n (P)

First

Let's say we have P and it doesn't contain g_1.

Take P, replace earliest choice (in P) w g_1 to make P'.

Whatever we replaced g_1 w had to have at least covered S, and had to finish $\leq f_{g_1}$, therefore it's safe to replace and we've created new opt. sol'n P' w g_1.

c) optimal substructure.

- If we combine two optimal subproblems can create overall optimal soln.

greedy choice

opt. soln remaining problem (Q)

Contradiction:

Assume $g_1 + Q$ is not an optimal soln. That means it's possible to cover (S, F) in fewer act. than $|g_1 + Q|$

- We know g_1 is part of some opt. soln
- ∴ There must be a set of activities in (S', F) (R) that can be combined w g_1 to be optimal

↑ must be an optimal soln to subproblem (or else we'd could get fewer act overall)

∴ $R \equiv Q$ optimal soln

Homework 5 Solutions

ECE 345 Algorithms and Data Structures
Winter Semester, 2015

First, we argue that Set Cover is in NP, since given a collection of sets C, a certifier can efficiently check that C indeed contains at most k elements, and that the union of all sets listed in S' does include all elements from the ground set U.

We will now show that Vertex-Cover $\leq_p$ Set Cover. Given an instance of Vertex Cover (i.e. a graph $G = (V, E)$ and an integer j), we will construct an instance of the Set Cover problem. Let $U = E$. We will define n subsets of U as follows: label the vertices of G from 1 to n, and let S_i be the set of edges that incident to vertex i (the edges for which vertex i is an end-point). Note that $S_i \subseteq U$ for all i. Finally let $k = j$. This construction can be done in time that is polynomial in the size of the Vertex Cover instance. We now run our black box for the Set Cover problem and return the same result it gives.

To prove that this answer is correct, we simply need to show that the original instance of Vertex Cover is a yes instance if and only if the Set Cover instance we created is also a yes instance.

Suppose G has a vertex cover of size at most j. Let S be such a set of nodes. By our construction, S corresponds to a collection C of subsets of U. Since we defined $k = j$, C clearly has at most k subsets. Furthermore, we claim that the sets listed in C cover U. To see this, consider any element of U. Such an element is an edge e in G, and since S is a vertex cover for G, at least one of es endpoints is in S. Therefore C contains at least one of the sets associated with the endpoints of e, and by definition, these both contain e.

Now suppose there is a set cover C of size k in our constructed instance. Since each set in C is naturally associated with a vertex in G, let S be the set of these vertices. $|S| = |C|$ and thus S contains at most j nodes. Furthermore, consider any edge e. Since e is in the ground set U and C is a set cover, C must contain at least one set that includes e. But by our construction, the only sets that include e correspond to nodes that are endpoints of e. Thus, S must contain at least one of the endpoints of e. We have now shown that our algorithm solves the Vertex Cover problem using a black box for the Set Cover problem. Since our contruction takes polynomial time, and we have shown that Set Cover is in NP, we can conclude that Set Cover is NP-Complete.

2. H-FINAL $\in$ NP because given a schedule we can easily check that no student has two exams in the same time slot.

Now we have to show that H-COLOR $\leq_q$ H-FINAL. For input G to H-COLOR, let each vertex correspond to an exam. Let each edge correspond to a student and his/her two assigned exams.

$\Rightarrow$ If $G \in$ H-COLOR, then we can produce a valid schedule. The color of the vertex corresponds to the time the exam is scheduled. Since no two vertices connected by an edge can be colored the same, the two exams that each student must take must be scheduled for different times.

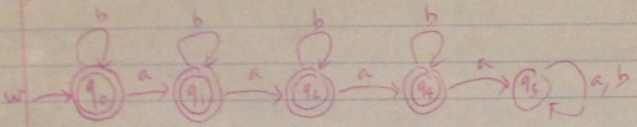
$\Leftarrow$ If there is a valid schedule in H time slots, we can produce a H-COLORing of G. For all exams in the same time slot, color the corresponding vertices the same. Pick a different color

for each time slot (we have H slots and H colors). Any 2 connected vertices correspond two assigned exams of a student, which cannot be scheduled for the same time in a valid schedule and hence, the two vertices must have 2 different colors.

3. See the attachment.

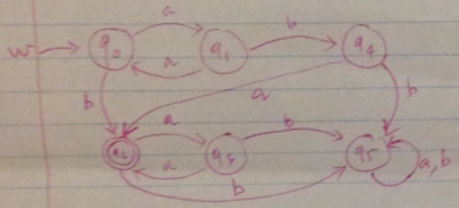
a) $b^* + b^*ab^* + b^*ab^*ab^* + b^*ab^*ab^*ab^*$

a) $b^* + b^*ab^* + b^*ab^*ab^* + b^*ab^*ab^*ab^*$



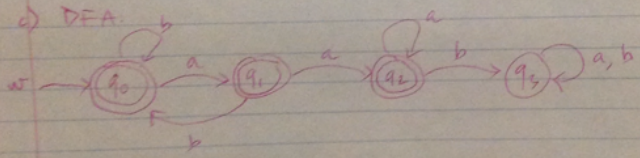
accepting states: q_0, q_1, q_2, q_4

b) $(aa)^*b(aa)^* + (aa)^*aba(aa)^*$



accepting state: q_2

c) DFA:



accepting states q_0, q_1, q_2

RE $(b+ab)^*a^*$ (No mark, optional)

"These are potential solutions. All correct answers are accepted."

Homework 3 Solutions

ECE 345 Algorithms and Data Structures
Fall Semester, 2013

1. Inorder traversal: left children of a binary tree is visited first, then its root, and finally its right children. Let us call the given node as a. There are two cases to consider in this problem:

   If a has a right child, the next node is the most left child in its right subtree. To find the next node in inorder traversal, go to the right child of a and traverse along the left child as much as possible.

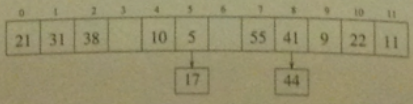
   If a does not have a right child, its next node is the first ancestor node b such that b goes to its left child and keep going right to reach a.

2. Recursive algorithm:

   isBST(node)
   if ($node.left == NULL$ and $node.right == NULL$)
   return $(true, node.data, node.data)$;
   $maxLeft$=$node.data$;
   if ($node.left != NULL$)
   $(isLeftBST, minLeft, maxLeft)$ = isBST($node.left$);
   if ($isLeftBST == false$ or $maxLeft >$ $node.data$)
   return $false$;
   $maxRight$ = $node.data$;
   if ($node.right != NULL$)
   $(isRightBST, minRight, maxRight)$ = isBST($node.right$);
   if ($isRightBST == false$ or $minRight <$ $node.data$)
   return $false$;
   return $(true, minLeft, maxRight)$;

   The runtime is $O(n)$.

3. The final hash table is:



4. Say n boxes arrive in the order $b_1, \ldots, b_n$. Say each box b_i has a positive weight w_i, and the maximum weight each truck can carry is W. To pack the boxes into N trucks *preserving the order* is to assign each box to one of the trucks $1 \ldots N$ so that:

- No truck is overloaded: the total weight of all voxes in each truck is less than or equal to W.
- The order of arricals is preserved: if the box b_i is sent before the box b_j (i.e. box b_i is assigned to truck x, box b_j is assigned to truck y, and $x < y$), then it must be the case that b_i has arriced to the company earlier than b_j (i.e. $i < j$).

We prove that the greedy algorithm uses the fewest possible trucks by showing that it "stays ahead" of any other solution. Specifically, we consider any other solution and show the following. If the greedy algorithm fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution fits $b_1, \ldots, b_i$ into the first k trucks, then $i \leq j$. Note that this implies the optimality of the greedy algorithm, by setting k to be the number of trucks used by the greedy algorithm.

We will prove this claim by induction on k. The case $k = 1$ is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now, assuming it holds for $k - 1$: the greedy algorithm fits j' boxes into the first $k - 1$, and the other solution first $i' \leq j'$. Now, for the k^{th} truck, the alternate solution packs in $b_{i'+1}, \ldots, b_i$. Thus, since $j' \geq i'$, the greedy algorithm is able at least to fit all the boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and it can potentially fit more. This completes the induction step, the proof of the claim, and hence the proof of optimality of the greedy algorithm.

5. Define an array $t[]$ such that $t[i] = true$ if $s[0 \cdots i-1]$ can be segmented into words. Initial state $t[0] = true$.

```
isWords(s, dict)
    t[0] = true ;
    for i = 1 to s.size()
        if s[0..i − 1] in dict
            t[i] = true;
            continue;
        j = 0;
        while !(s[j..i − 1] in dict and t[j])
            j++ ;
        if (s[j..i − 1] in dict and t[j])
            t[i] = true ;
    return t[s.size];
```

The run-time is $O(n^2)$ and memory usage is $O(n)$.

6. Let $OPT(i)$ denote the minimum cost of a solution for weeks 1 through i. In an optimal solution, we either use company A or company B for the ith week. If we use company A, we pay rs_i and can behave optimally up through week $i - 1$. If we use company B for week i, then we pay $4c$ for this contract, and so there's no reason not to get the full benefiy of it by starting at week $i - 3$; thus we can behave optimally up through week $i - 4$, and then invoke this contract.

Thus we have:

$OPT(i) = min(rs_i + OPT(i-1), 4c + OPT(i-4))$.

We can build up these OPT values in order of increasing i, spending contant time per iteration, with the initialization $OPT(i) = 0 for i \leq 0$.

The final answer is $OPT(n)$ and we can obtian the schedule by tracing back through the array of OPT values.

Homework 4 Solutions

ECE 345 Algorithms and Data Structures
Winter Semester, 2015

1. (a) We want to find the path p with maxim al strength $S(p)$ from u to v. This is equivalent t maximizing $\log S(p) = \sum \log(I(u_i, u_{i+1}))$ which is in turn equivalent to minimizing $(p) = -\log S(p) = \sum \log \frac{1}{I(u_i,u_{i+1})}$. o for each edge (u, v), we can assign its weight a $\log \frac{1}{I(u_i,u_{i+1})}$ which is non negative as $0 < I(u_i, u_{i+1}) < 1$. As a result, the problem is c nverted to finding the shortest path f om u to v.

   (b) T enforce k friends between u and v, e modify Bellman-Ford algorithm. During each it ration, we first check which edges ne d to be relaxed, but only update the distances a er checking all the edges. This ensur s that relaxation is only based on the distances fr m the previous iteration. This means after iteration i, $v.d = min\{w(p) : |p| \leq i\}$.

```
K ength-Bellman-Ford(G, s, k)
fo each node v ∈ G.V
   v.d = ∞
s. = 0
fo i = 1 to k
   T =
   for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
         T[v] = u.d + w(u, v)
   for each v ∈ T
      v.d = T[v]
```

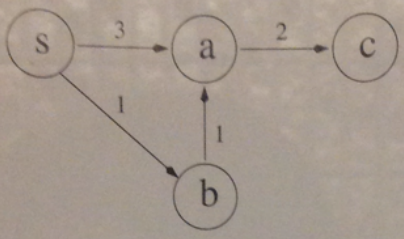
In ialization takes $O(V)$ time. Both ch cking and updating take $O(E)$ time and there ar k iterations. So the total time is $O(k \cdot E + V)$.

No e that simply using Bellman-Ford w th k iterations does not work because certain ed relaxation orders will find the shor est paths with length greater than k in fewer th k iterations. Consider the graph in igure 1, suppose we want to find shortest paths wi length at most 2 from s. If we rela the edges in the order $(s, b), (b, a), (s, a), (a, c)$ aft the first iteration the shortest path o c has length 3.

Si larly, using BFS with k iterations do s not work either. Considering the same exam- pl in the second iteration, b and a are ir the frontier. If we check b's neighbors first and up ate a with weight 2. When we check 's neighbors, we update c with weight 4 but for thi weight the shortest path has length 3. The key is we need to use the distance values fro previous iteration instead of the new values in the current iteration for update.

Si ly modifying Bellman-Ford to keep rack of the length of the current path from s to and avoid relaxation when length g ater than k also does not work. Consider the gra h in Figure 1 and we want to find th shortest path with length 2. If we relax the ed s in the order: $(s, b), (b, a), (s, a), (a,)$ and keep track of the path lengths, we will not find the path $s- > a- > c$. When we try to relax edge (a, c), the shortest path from

s to a has length 2 and hence the algorithm would avoid update $c.d$ as its shortest path has length 3. $t.d$ remains to be ∞.



2. (a) Consider the graph with 3 vertexes a, b, c and $w(a \to b) = 1$, $w(a \to c) = 2$, $w(c \to b) = -3$. Dijkstra will find that the shortest path from a to b is 1 but it is actually -1 $(a \to c \to b)$.

   (b) High level: As original Bellman-Ford can detect a negative weight cycle that is reachable from the source s, we just have to add a vertex s' so that any negative weight cycle if existed is reachable from s'.

   We construct graph G' from the original graph $G = (V, E)$ by adding a new vertex s' and for each vertex $v \in V$, add one new directed edge (s', v) to the graph G and let the edge weight equal 1.

   Since the newly added vertex s' has a directed edge to each vertex of the graph G, Bellman-Ford algorithm from s' finds any negative weight cycle in G if existed.

   The new graph $G' = (V', E')$ has $|V'| = |V| + 2 = \mathcal{O}(|V|)$ and $|E'| = |E| + 2|V|$. The time complexity of Bellman-Ford on G' is bounded by $\mathcal{O}(|V'||E'|) = \mathcal{O}(|V||E| + |V||V|)$.

3. (a) We will prove that a minimum weight spanning tree under the normal definition, is also a spanning tree with the minimum value of the largest edge weight.

   Consider an MST T, suppose there exists a spanning tree T' such that the largest edge weight is smaller than the largest edge weight in T. Call the corresponding edges, e' and e respectively. Remove e from T. This breaks T into two connected components. There must exist an edge e'' in T' that connects these components. Clearly, $w(e'') < w(e') < w(e)$. Thus the tree T'' obtained from T by replacing the edge e with e'' has weight $w(T'') = w(T) + w(e'') - w(e) < w(T)$, which is a contradiction. We can use Prim's algorithm to solve the problem in time $O(VlogV + E)$.

   (b) Let (u, v) be the edge not in T whose weight decreased. Using DFS or BFS, find the unique simple path from u to v in T. Find an edge e of maximal weight on that path. If the weight of e is greater than that of (u, v), replace e in T with (u, v).

4. Call STRONGLY-CONNECTED-COMPONENTS (CLRS 3rd, p. 615) and form the component graph. Topologically sort the component graph. It is possible to topologically sort the component graph, because component graph is always a DAG. Verify that the sequence of vertexes $(v_1, v_2, \cdots, v_k)$ given by topological sort forms a linear chain in the component graph.

$OPT(i) = min(rs_i + OPT(i-1), 4c + OPT(i-4))$.

We can build up these OPT values in order of increasing i, spending contant time per iteration, with the initialization $OPT(i) = 0 for i \leq 0$.

The final answer is $OPT(n)$ and we can obtian the schedule [illegible]

That is, verify t at the edges $(v_1, v_2), (v_2, v_3), \cdots, (v_{k-1}, v_k)$ exist in the component graph. If the vertexes forn a linear chain, then the original grap is semi-connected; otherwise it is not.

Because we kno that all vertexes in each SCC are r utually reachable from each other, it suffices to show t at the component graph is semi-conn ected if and only if it contains a linear chain. Total runn ng time is $\Theta(V + E)$.