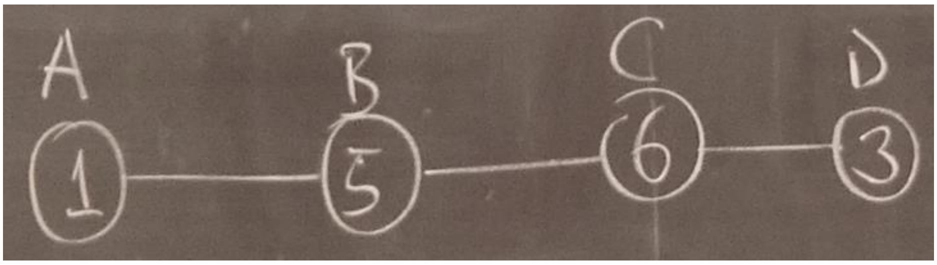
Linear Independent Set

February 14, 2019 2:33 PM

Input: An undirected line graph $G(V, E)$ and (non-negative) weights on nodes, where $n = |V|$



Output: The max-weight independent set in G.

i.e. a subset $S^* \subseteq V$ such that $\forall u, v \in S^*$ u and v are not connected

e.g. $\{A, C\} = 7$, $\{A, D\} = 4$, $\{B, D\} = 8$
output would be $\{B, D\}$

Naïve Algorithm: Search all 2^n subsets of V and keep track of the max weight independent subset

Runtime: $\Theta(2^n)$

Greedy Algorithm: Pick vertex v with highest weight; put v into output; remove neighbours of v.

This gives an output of $\{A, C\}$ $\Rightarrow$ which is incorrect

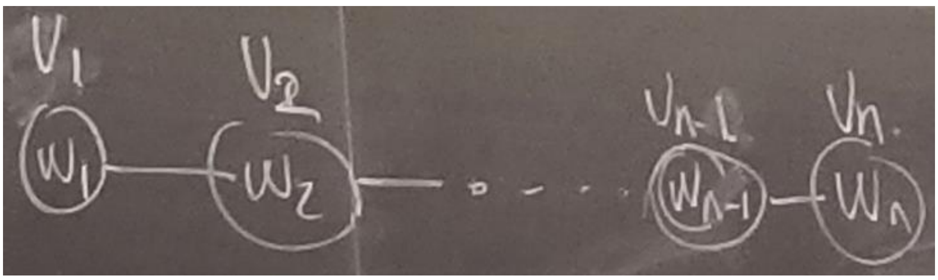
Divide and Conquer Approach: Divide and recurse into left and right.

$\Rightarrow$ need to deal with conficts at boundaries
$\Rightarrow$ can be made to work but will be slow

A new Dynamic Programming Algorithm:

Let S^* be the optimal solution.

Reason about what S^* looks like in terms of optimal solutions to smaller subproblems



A Simple Claim (Tautology): Only 2 possibilities:

Case 1: $v_n \notin S^*$

Claim 1: Then S^* is opt_{n-1}

Proof:

$G_n = G$
$G_{n-1} = G - \{v_n\}$
$G_{n-2} = G - \{v_n, v_{n-1}\}$
$\vdots$
$G_i = G - \{v_n v_{n-1}, \dots, v_{i+1}\}$

Claim 1: Then S^* is opt_{n-1}

Proof:

Assume for a contradiction that there exists S' solution in G_{n-1} where

$W(S') > W(S^*)$

Then, since S' is a linear independent set in G_n, then S^* can be opt_n

$\vdots$

$G_i = G - \{v_n v_{n-1}, \ldots, v_{i+1}\}$

$opt_n = S^*$
$opt_{n-1} =$ optimal solution to G_{n-1}
$\vdots$
$opt_i =$ optimal solution fo G_i

Case 2: $v_n \in S^*$

only contains vertices from $v_1, \ldots, v_{n-2}$

Claim 2: Then $S^* - \{v_n\}$ is opt_{n-2}

Proof:

Note $S^* - \{v_n\}$ only contains nodes $v_1, \ldots, v_{n-2}$ because $v_{n-1} \notin S^*$ would not be linearly independent.

Assume for a contradiction that there exists S'' in G_{n-2} that is linearly independent and $W(S'') > W(S^*)$

Then, $S'' \cup \{v_n\}$ is a linearly independent set in G_n with a higher weight than S^*, contradcition that S^* is opt_n

Therefore,

If $v_n \notin S^* \Rightarrow S^*$ is opt_{n-1}
If $v_n \in S^* \Rightarrow S^* - \{v_n\}$ is opt_{n-2}

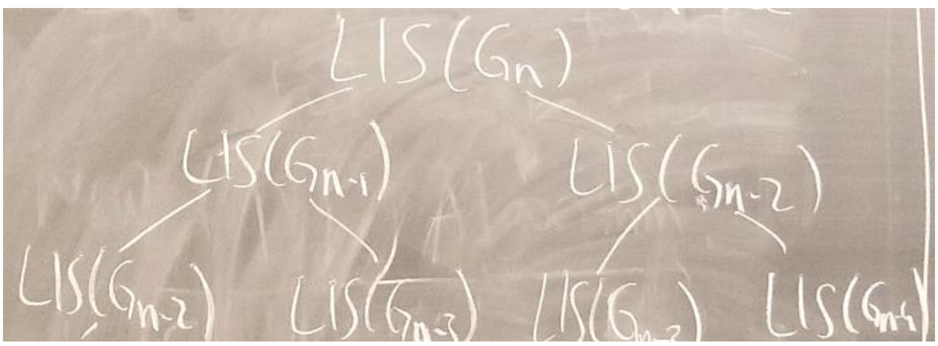
A possible recursive algorithm:

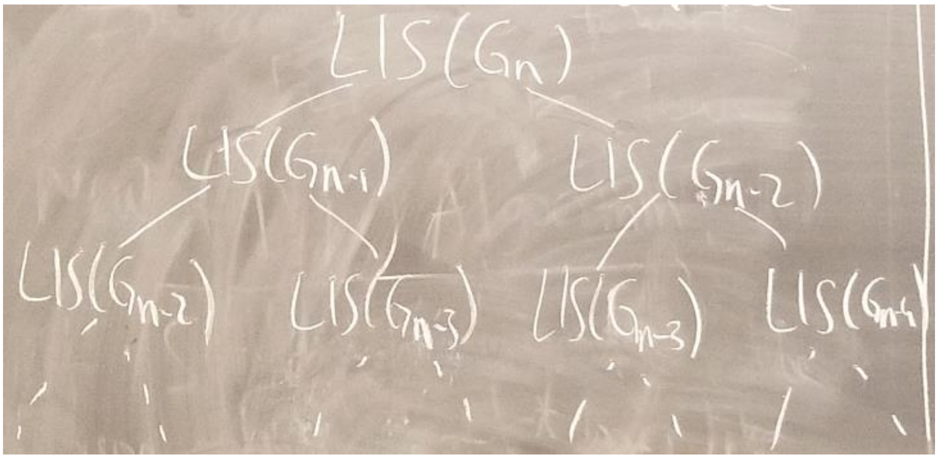
```
Rec-LIS( G(V,E), weights ) {
    Base Case:   if |V| == 1;   return ...
    s1 = Rec-LIS( G_{n-1} )
    s2 = Rec-LIS( G_{n-2} ) ∪ { v_n }
    return max(s1, s2)
}
```

Runtime: $T(n) = T(n-1) + T(n-2) + O(1) \Rightarrow \Theta(2^n)$

Exercise: Prove by induction

Let's look at the recursion tree:





Work is redundantly being repeated

Question: What is the # of distinct subproblems?

Answer: n for each G_i, there is one distinct subproblem

Fix 1: Memoization: Simply store or cache the results of each subproblem in a table. Before recursing, check if it has been solved before.

Exercise: Show that with memoization, runtime becomes $O(n)$

** This is not actually dynamic programming **

Fix 2: Bottom-up Iterative solution.

** Dynamic Programming **

A[n]: a solution array of length n

Tip: Write what each cell means in english

// A[i]: $opt_i \Rightarrow$ max weight linearly independent set in G_i

Dynamic Programming Algorithm:

Base Cases: A[0] = 0; A[1] = w_1

```
for i = 2…n {
        A[i] = max{A[i-1], A[i-2]+w_i}
}
```

return A[n]

Correctness: Follows from correctness of recurrence A[i] = max{A[i-1], A[i-2]+w_i}

A more formal proof would inductively prove A[i] = opt_i

Runtime: $O(n)$

Space: $O(n)$ but can be done in $O(1)$ by only keeping track of A[i-1], A[i-2]

Reconstructing the Solution:

Option 1: For each $A[i]$, also store the solution as we fill $A[i]$ $\Rightarrow$ $O(n^2)$ space

Option 2: Backtracing

Observe: $v_i \in opt_i$ if and only if $opt_{i-2} > opt_{i-1}$, $A[i-2] + w_i > A[i-2]$

Ex.

$v_n \in ?\ S^*$ $\Rightarrow$ $v_n \notin S^*$ $A[n-2] + w_n > A[n-1]$
$v_{n-1} \in ?\ S^*$ $\Rightarrow$ $v_{n-1} \in S^*$ $A[n-3] + w_{n-1} > A[n-2]$

$v_{n-3} \in ?\ S^*$ ...

```
Reconstruct ( G(V, E), weights ) {
    A = DP-LIS(G, weights)
    i = n;  S* = Ø
    while ( i > 0 ) {
        if ( A[i-2]+w_i > A[i-1] ) {
            S*.add( v_i )
            i = i-2
        }
        else {
            i = i - 1
        }
    }
    return S*
}
```

Runtime: $O(n)$