CS 161: Design and Analysis of Algorithms

Dynamic Programming I: Sequence Alignment/Edit Distance

- Definition
- Algorithm
- Underlying dag
- Variants
- Other problems

String Distances

- Want to find strings that are “close” to each other
  - Example: spellcheckers want to find a word that is close by to a misspelled word
  - Example: find DNA sequences that are similar
- What is a good measure of closeness?

Sequence Alignment

```
S – N O W Y
S U N N – Y
```

- A way of writing to strings next to each other, showing edits

Sequence Alignment

- Common Operations
  - Insertions: insert a character into a string
  - Deletions: delete a character from a string
  - Substitutions: replace character with another
  - Swap: swap adjacent characters
- Many possible types of alignments based on which operations we choose to consider

Edit Distance

- Associate a cost to every alignment
- Possibilities:
  - All operations have cost 1
  - Different operations have different costs
  - Cost depends on characters involved
- Edit distance = minimum cost over all possible alignments
- Optimal sequence alignment = alignment with minimum cost

Levenshtein Distance

- Operations: insertions, deletions, substitutions
- Cost: 1 per operation

```
S – N O W Y
S U N N – Y
```

Cost: 3

Dynamic Programming Solution

- Suppose we have an optimal alignment A.
- Look at the characters of each string in the last position. Either:
  - They are the same
  - They are different
  - One is a ‘ – ’

Dynamic Program Solution

- Comparing strings S and T
- Let E(i,j) be the cost of the solution for the first i characters of S and the first j characters of j
- Objective: compute E(|S|,|T|)

Dynamic Programming Solution

- Consider rightmost column of solution E(i,j).
- Can only be three things:

	S_i	S_i	–
	T_j	–	T_j
Cost:	0 or 1	1	1
Must align:	E(i-1,j-1)	E(i-1,j)	E(i,j-1)

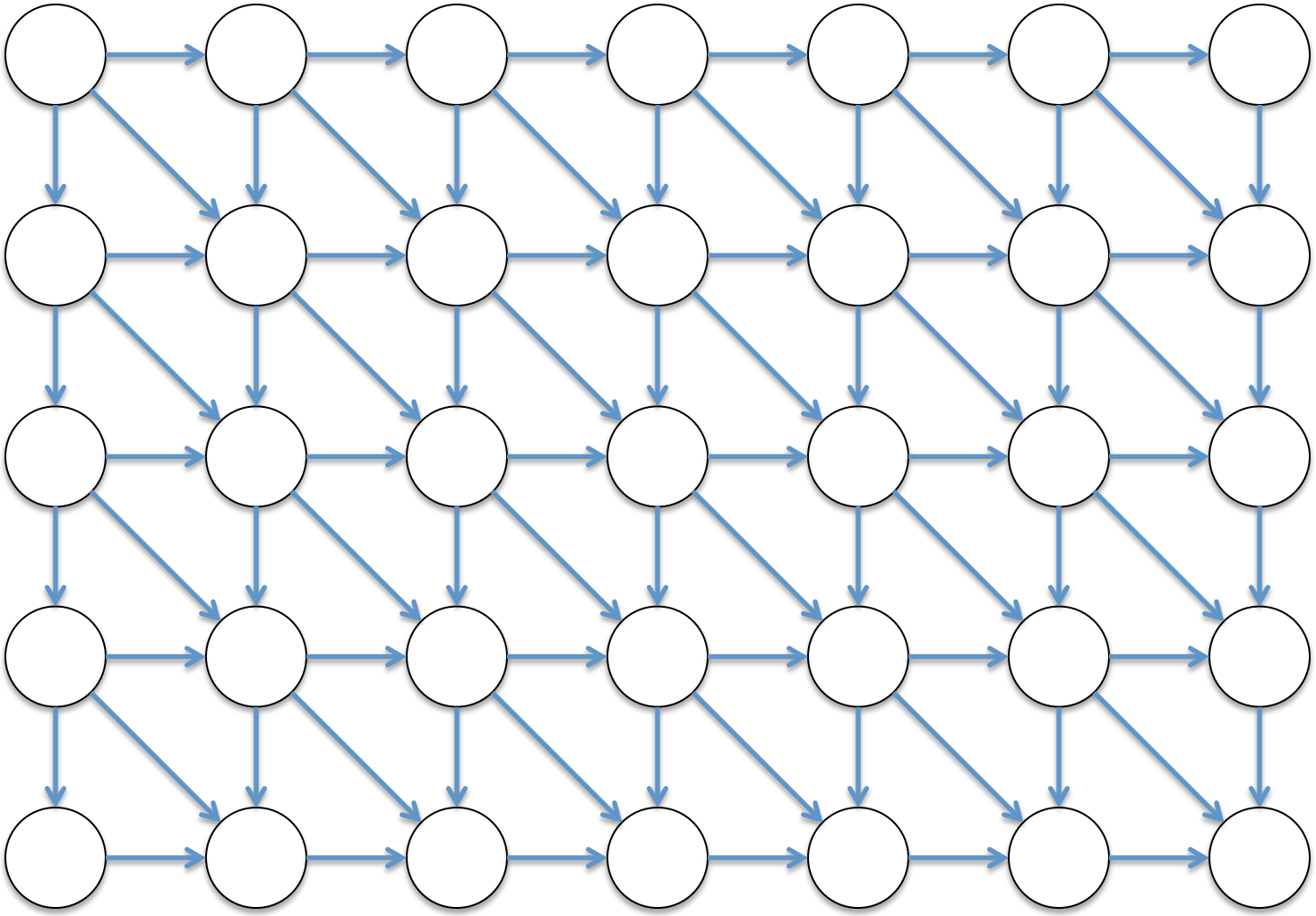
Dynamic Programming Solution

- Let diff(i,j) = 0 if $S_i = T_j$, 1 otherwise
- E(i,j) = min{diff(i,j) + E(i-1,j-1),
  1 + E(i-1,j),
  1 + E(i,j-1)}

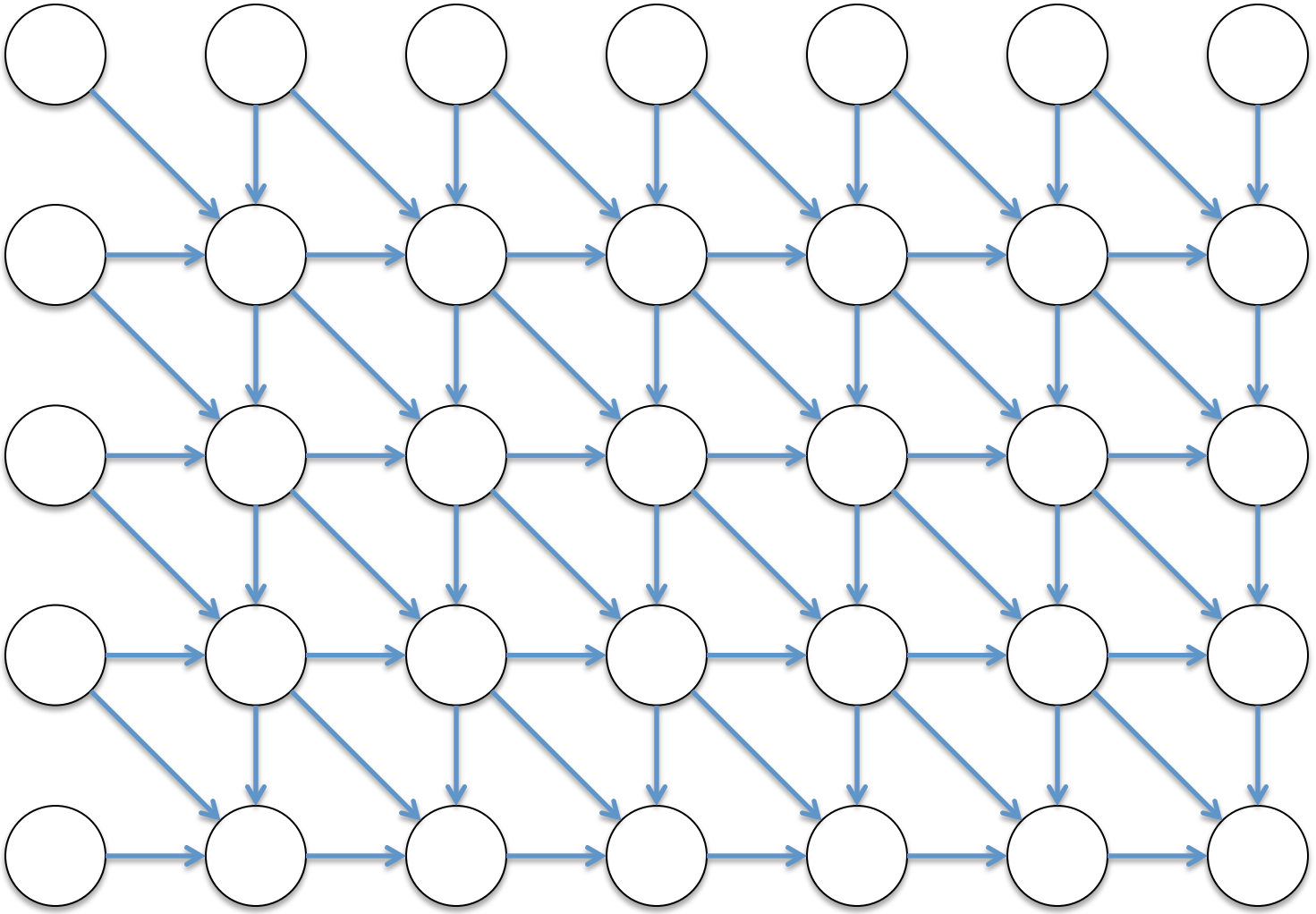
Dynamic Programming Solution

- Base cases: E(i,0) = I, E(0,j) = j
- Running Time:
  - Computing solution to each subproblem takes constant time
  - |S| * |T| subproblems
  - O(|S| |T|)

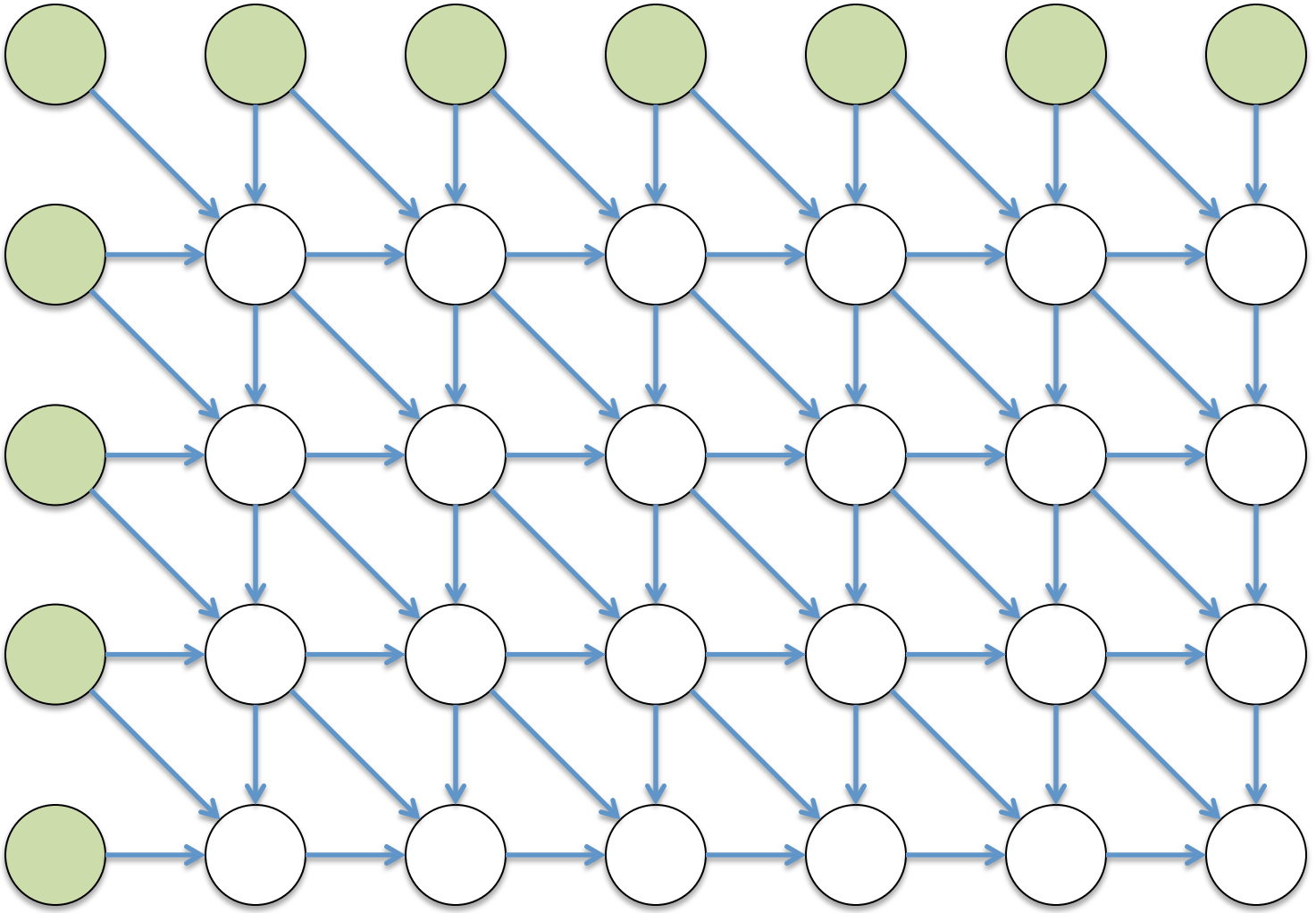
Underlying Dag



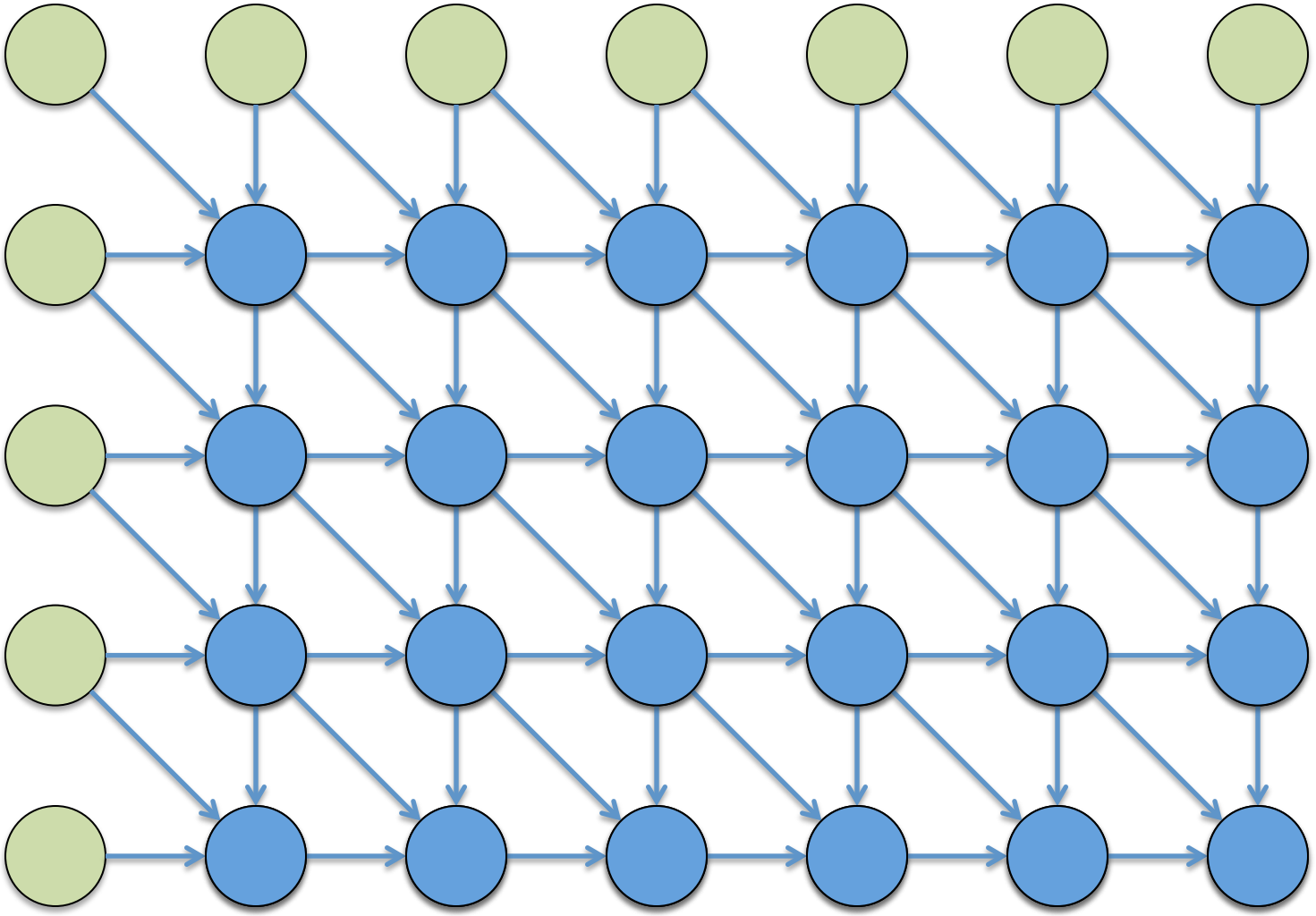
Underlying Dag



Underlying Dag



Underlying Dag



Algorithm

For i = 0, 1, …, |S|: E(i,0) = i

For j = 0, 1, …, |T|: E(0,j) = j

For i = 0, 1, …, |S|:

    For j = 0, 1, …, |T|:

        E(i,j) = min{diff(i,j) + E(i-1,j-1),

                     1 + E(i-1,j),

                     1 + E(i,j-1)}

Return E(|S|,|T|)

Finding Optimal Alignment

- To find the actual optimal sequence alignment, need to store partial solutions as well
- A(i,j) = optimal sequence alignment for first i characters of S, first j characters of t

Finding Optimal Alignment

- Perform checks:
  - If diff(i,j) + E(i-1,j-1) is minimum, A(i,j) is the alignment A(i-1,j-1), adding a last column consisting of the last letter of S and T
  - If 1 + E(i-1,j) is minimum, A(i,j) is A(i-1,j), adding a last column consisting of the last letter of S, and a dash for T
  - Similar for 1 + E(i,j-1) being minimum

Example

- S = “FOR”, T = “IF”
- E(0,0) = 0, A(0,0) = (“”, “”)
- E(1,0) = 1, A(1,0) = (“F”, “–”)
- E(2,0) = 2, A(2,0) = (“FO”, “– –”)
- E(3,0) = 3, A(3,0) = (“FOR”, “– – –”)
- E(0,1) = 1, A(0,1) = (“–”, “I”)
- E(0,2) = 2, A(0,2) = (“– –”, “IF”)

Example

- S = “FOR”, T = “IF”
- E(1,1):
  - diff(i,j) + E(i-1,j-1) = 1 + 0 = 1
  - 1 + E(i-1,j) = 1 + 1 = 2
  - 1 + E(i,j-1) = 1 + 1 = 2
  - Therefore, E(1,1) = 1, A(1,1) = (“F”, “I”)

Example

- S = “FOR”, T = “IF”
- E(1,2):
  - diff(i,j) + E(i-1,j-1) = 0 + 1 = 1
  - 1 + E(i-1,j) = 1 + 2 = 3
  - 1 + E(i,j-1) = 1 + 1 = 2
  - Therefore, E(1,2) = 1, A(1,2) = (“–F”, “IF”)

Example

- S = “FOR”, T = “IF”
- E(2,1):
  - diff(i,j) + E(i-1,j-1) = 1 + 1 = 2
  - 1 + E(i-1,j) = 1 + 1 = 2
  - 1 + E(i,j-1) = 1 + 2 = 3
  - Therefore, E(1,2) = 2, A(1,2) = (“FO”, “–I”)

Example

- S = “FOR”, T = “IF”
- E(2,2):
  - diff(i,j) + E(i-1,j-1) = 1 + 1 = 2
  - 1 + E(i-1,j) = 1 + 1 = 2
  - 1 + E(i,j-1) = 1 + 2 = 3
  - Therefore, E(2,2) = 2, A(2,2) = (“FO”, “IF”)

Example

- S = “FOR”, T = “IF”
- E(3,1):
  - diff(i,j) + E(i-1,j-1) = 1 + 2 = 3
  - 1 + E(i-1,j) = 1 + 2 = 3
  - 1 + E(i,j-1) = 1 + 3 = 4
  - Therefore, E(3,1) = 3, A(3,1) = (“FOR”, “– –I”)

Example

- S = “FOR”, T = “IF”
- E(3,2):
  - diff(i,j) + E(i-1,j-1) = 1 + 2 = 3
  - 1 + E(i-1,j) = 1 + 2 = 3
  - 1 + E(i,j-1) = 1 + 3 = 4
  - Therefore, E(3,1) = 3, A(3,1) = (“FOR”, “–IF”)

Variants

- Easy to modify algorithm to handle variants
- Example: no replacements
  - $E(i,j) = \min\{1 + E(i-1,j),$
    $1 + E(i,j-1)\}$

Variants

- Weighted Operations
  - Insertions/deletions get cost d
  - Replacing x with x' get cost C(x,x')
  - $E(i,j) = \min\{C(S_i,T_j) + E(i-1,j-1),$
    $d + E(i-1,j),$
    $d + E(i,j-1)\}$

Longest Increasing Subsequence

- Given sequence of numbers (a_1, …, a_n)
- A subsequence is a subset taken in order
  - Ex: (a_2, a_3, a_6, a_{10}, …)
- An increasing subsequence is one where numbers get strictly larger
  - Ex: $a_2 = 3$, $a_3 = 7$, $a_6 = 9$, $a_{10} = 12$, …
- Goal: find longest increasing subsequence

Longest Increasing Subsequence

- Let E(i) be length of longest increasing subsequence of (a_1, …, a_i)
- Either longest sequence includes a_i, or it doesn't
  - If it does, length of longest sequence is E(j) + 1 for some j with $a_j < a_i$
  - Otherwise, E(i-1)

Longest Increasing Subsequence

- Algorithm:

  E(0) = 1

  For i = 1, …, n:

    E(i) = max{E(i-1),

      1+E(j) for j such that $a_j < a_i$}

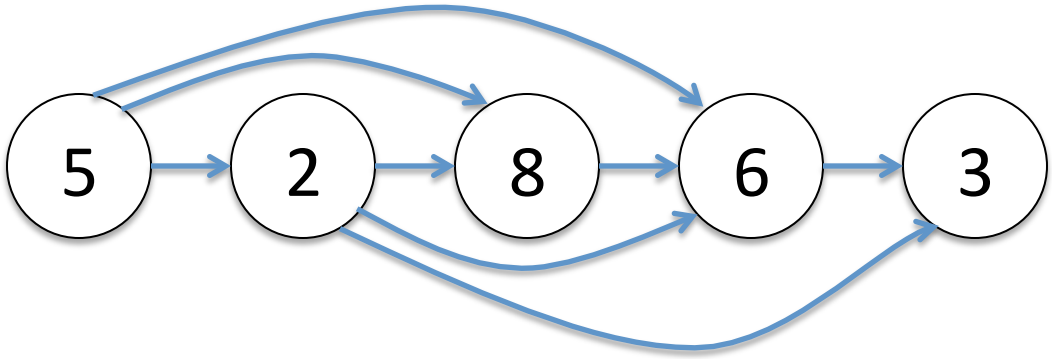
  Return E(n)

Longest Increasing Subsequence

- Running Time?
  - For each E(i), need to minimize over potentially all E(j) for j<i
  - Running time $O(n^2)$

Underlying Dag

- Arrow from j to i if $a_j < a_i$, or $j = i - 1$
- Example: (5,2,8,6,3)

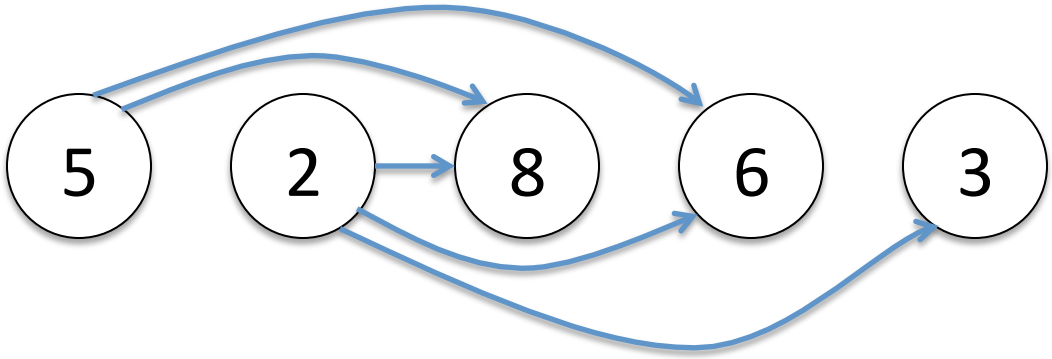


An Alternate Approach

- Let F(i) be the length of the longest increasing subsequence **ending** with a_i
- F(i) = 1 + max(L(j) for $a_j < a_i$)

Underlying Dag

- Arrow from j to i if $a_j < a_i$
- Example: (5,2,8,6,3)

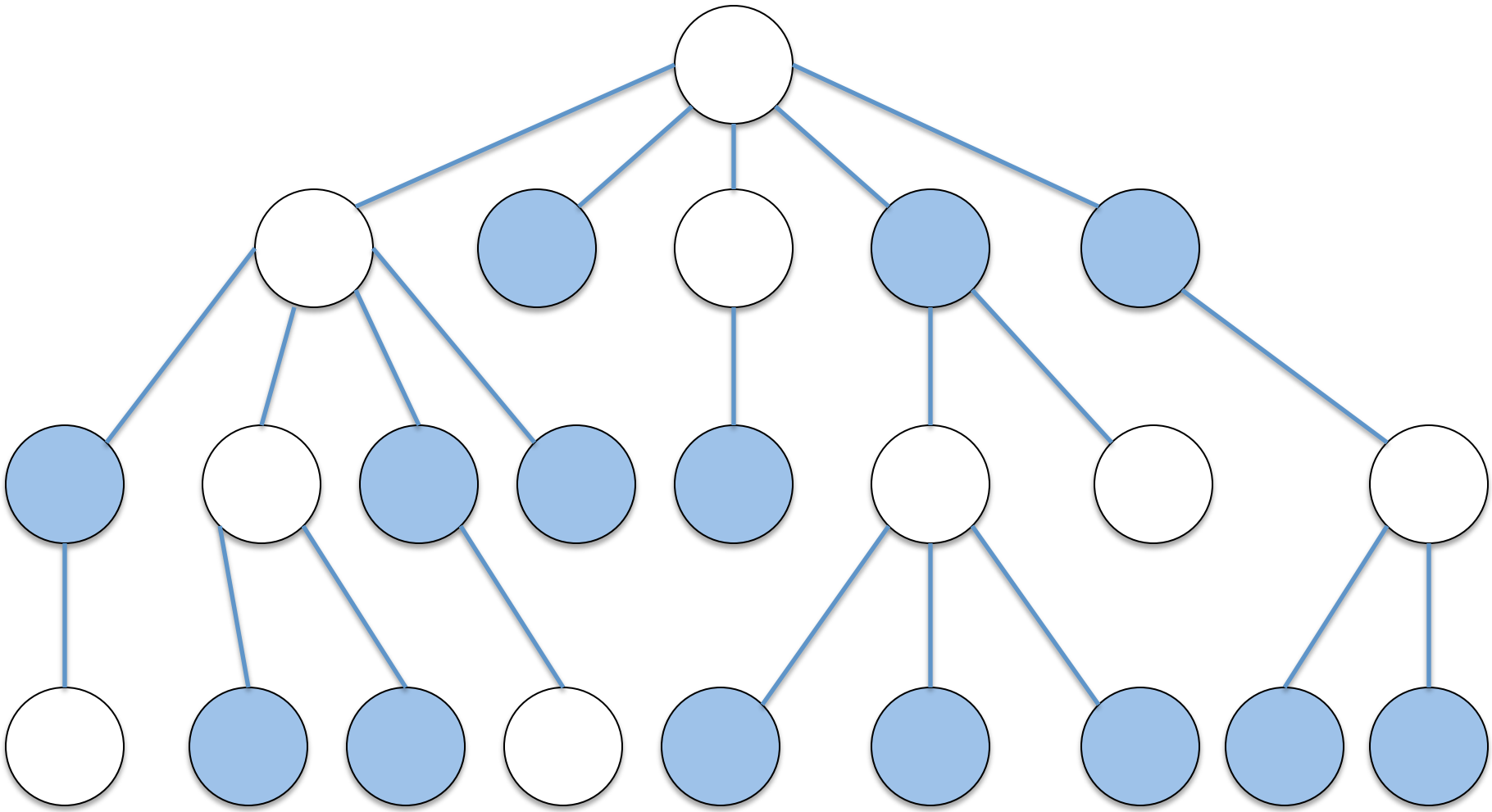


- Solution = longest path in dag

Independent Sets In Trees

- Given a graph G = (V,E), a subset of nodes S in independent if there are no edges between nodes of S
- Goal: find largest independent set
- In general, very hard problem
- Special case: Trees

Independent Sets in Trees



Independent Sets in Trees

- F(v): maximal independent set for subtree rooted at v
- Either F(v) contains v, or it doesn't
  - If it does, F(v) = 1 + Sum(F(u): u grandchild of v)
  - Otherwise, F(v) = Sum(F(u): u child of v)
- Base case: leaves get F(v) = 1
- Work way up to root
- Underlying dag: tree with edges pointing to parent

Algorithm

- Each node has two values:
  - F(v): size of maximum independent set
  - C(v): sum of F(v) values for children
- C(v) = Sum(F(u): u child of v)
- F(v) = max(1 + Sum(C(u): u child of v), C(v))
- Running time: O(|V| + |E|)

Midterm Statistics

- Average: 126/200
- Standard Deviation: 35