

# CSE 548: (Design and) Analysis of Algorithms

## Dynamic Programming

R. Sekar

1/56

## Overview

- Another approach for *optimization problems*, more general and versatile than greedy algorithms.
- *Optimal substructure* The optimal solution contains optimal solutions to subproblems.
- *Overlapping subproblems*. Typically, the same subproblems are solved repeatedly.
- Solve subproblems in *a certain order*, and *remember solutions* for later reuse.

2/56

## Topics

- |                              |                               |
|------------------------------|-------------------------------|
| 1. Intro                     | UNIX apps                     |
| Overview                     |                               |
| Topological Sort             |                               |
| DAGs and Dynamic Programming |                               |
| 2. LIS                       | 4. Knapsack                   |
| DAG Formulation              | Knapsack w/ Repetition        |
| Algorithm                    | 0-1 Knapsack                  |
| 3. LCS                       | 5. Chain MM                   |
| Defn                         | Memoization                   |
| Towards Soln.                | 6. Fixpoints & Shortest Paths |
| Variations                   | Iterative Solving             |
| Seq. Alignment               | Shortest Path                 |
|                              | SSP                           |
|                              | ASP I                         |
|                              | ASP II                        |

3/56

## Topological Sort

A way to linearize DAGs while ensuring that for every vertex, all its ancestors appear before itself.

**Applications:** Instruction scheduling, spreadsheet recomputation of formulas, Make (and other compile/build systems) and Task scheduling/project management.

**Captures dependencies**, and hence arises frequently in all types of programming tasks, and of course, dynamic programming!

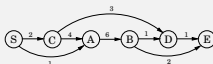
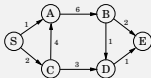
4/56

## Topological Sort

**topoSort**( $V, E$ )

```

while  $|V| \neq 0$ 
  if there is a vertex  $v$  in  $V$  with in-degree of 0
    output  $v$ 
     $V = V - \{v\}; E = E - \{e \in E \mid e \text{ is incident on } v\}$ 
  else output "graph is cyclic"; break
return
    
```



5/56

## Topological Sort

**topoSort**( $V, E$ )

```

while  $|V| \neq 0$ 
  if there is a vertex  $v$  in  $V$  with in-degree of 0
    output  $v$ 
     $V = V - \{v\}; E = E - \{e \in E \mid e \text{ is incident on } v\}$ 
  else output "graph is cyclic"; break
return
    
```

**Correctness:**

- If there is no vertex with in-degree 0, it is not a DAG
- When the algorithm outputs  $v$ , it has already output  $v$ 's ancestors

**Performance:** What is the runtime? Can it be improved?

6/56

## Shortest paths in DAGs

**SSPDag**( $V, E, w, s$ )

```

for  $u$  in  $V$  do
   $\text{dist}(u) = \infty$ 
 $\text{dist}(s) = 0$ 
for  $v \in V - \{s\}$  in topological order do
   $\text{dist}(v) = \min_{(u,v) \in E} (\text{dist}(u) + w(u, v))$ 
    
```

*That's all!*

7/56

## DAGs and Dynamic Programming

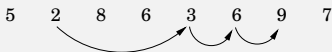
- *Canonical way to represent dynamic programming*
  - Nodes** in the DAG represent subproblems
  - Edges** capture dependencies between subproblems
  - Topological sorting** solves subproblems in the right order
  - Remember** subproblem solutions to avoid recomputation
- Many bottom-up computations on trees/dags *are* instances of dynamic programming
  - applies to trees of recursive calls (w/ duplication), e.g., Fib
- For problems in other domains, DAGs are implicit, and topological sort is also done implicitly
  - Can you think of a way to do topological sorting implicitly, without modifying the dag at all?

8/56

## Longest Increasing Subsequence

### Definition

Given a sequence  $a_1, a_2, \dots, a_n$ , its LIS is a sequence  $a_{i_1}, a_{i_2}, \dots, a_{i_k}$  that maximizes  $k$  subject to  $i_j < i_{j+1}$  and  $a_{i_j} \leq a_{i_{j+1}}$ .



9/56

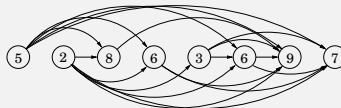
## Casting LIS problem using a DAG

**Nodes:** represent elements in the sequence

**Edges:** connect an element to all followers that are larger

**Topological sorting:** sequence already topologically sorted

**Remember:** Using an array  $L[1..n]$



10/56

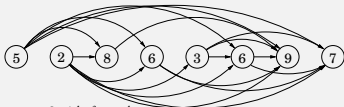
## Algorithm for LIS

$LIS(E)$

**for**  $j = 1$  **to**  $n$  **do**

$L[j] = 1 + \max_{(i,j) \in E} L[i]$

**return**  $\max_{j=1}^n L[j]$



**Correctness:** Straight-forward

**Complexity:** What is it? Can it be improved?

11/56

## Key step in Dyn. Prog.: Identifying subproblems

i. The input is  $x_1, x_2, \dots, x_n$  and a subproblem is  $x_1, x_2, \dots, x_i$ .

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10}$

The number of subproblems is therefore linear.

ii. The input is  $x_1, \dots, x_n$ , and  $y_1, \dots, y_m$ . A subproblem is  $x_1, \dots, x_i$  and  $y_1, \dots, y_j$ .

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10}$

$y_1 \ y_2 \ y_3 \ y_4 \ y_5 \ y_6 \ y_7 \ y_8$

The number of subproblems is  $O(mn)$ .

iii. The input is  $x_1, \dots, x_n$  and a subproblem is  $x_i, x_{i+1}, \dots, x_j$ .

$x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9 \ x_{10}$

The number of subproblems is  $O(n^2)$ .

iv. The input is a rooted tree. A subproblem is a rooted subtree.

12/56

## Subsequence

### Definition

A sequence  $a[1..m]$  is a subsequence of  $b[1..n]$  occurring at position  $r$  if there exist  $i_1, \dots, i_k$  such that  $a[r..(r+l-1)] = b[i_1]b[i_2] \dots b[i_l]$ , where  $i_j < i_{j+1}$

The relative order of elements is preserved in a subsequence, but unlike a substring, the elements need not be contiguous.

*Example:* BDEFHJ is a subsequence of ABCDEFGHIJK

13 / 56

## Longest Common Subsequence

### Definition (LCS)

The LCS of two sequences  $x[1..m]$  and  $y[1..n]$  is the longest sequence  $z[1..k]$  that is a subsequence of both  $x$  and  $y$ .

*Example:* BEHJ is a common subsequence of ABCDEFGHIJKLM and AABBXE/HJZ

By aligning elements of  $z$  with the corresponding elements of  $x$  and  $y$ , we can compare  $x$  and  $y$

$x : P \ R \ O \ F \ - \ E \ S \ S \ O \ R$

$z : P \ R \ O \ F \ - \ E \ S \ - \ - \ R$

$y : P \ R \ O \ F \ F_{ins} \ E \ S \ -_{del} \ U_{sub} \ R$

to identify the *edit* operations (insert, delete, substitute) operations needed to map  $x$  to  $y$

14 / 56

## Edit (Levenshtein) distance

### Definition (ED)

Given sequences  $x$  and  $y$  and functions  $I$ ,  $D$  and  $S$  that associate costs with each insert, delete and substitute operations, what is the minimum cost of any the edit sequence that transforms  $x$  into  $y$ .

### Applications

- Spell correction (Levenshtein automata)
- diff
- In the context of version control, reconcile/merge concurrent updates by different users.
- DNA sequence alignment, evolutionary trees and other applications in computational biology

15 / 56

## Towards a dynamic programming solution (I)

What subproblems to consider?

- Just like the LIS problem, we proceed from left to right, i.e., compute  $L[j]$  as  $j$  goes from 1 to  $n$
- But there are two strings  $x$  and  $y$  for LCS, so the subproblems correspond to prefixes of both  $x$  and  $y$  — there are  $O(mn)$  such prefixes.

E	X	P	O	N	E	N	T	I	A	L
P	O	L	Y	N	O	M	I	A	L	

The subproblem above can be represented as  $E[7, 5]$ .  
 $E[i, j]$  represents the edit distance of  $x[1..i]$  and  $y[1..j]$

16 / 56

## Towards a dynamic programming solution (2)

For  $E[k, l]$ , consider the following possibilities:

- $x[k] = y[l]$ : in this case,  $E[k, l] = E[k-1, l-1]$  — the edit distance has not increased as we extend the string by one character, since these characters match
- $x[k] \neq y[l]$ : Three possibilities
  - extend  $E[k-1, l]$  by deleting  $x[k]$ :  $E[k, l] = E[k-1, l] + DC(x[k])$
  - extend  $E[k, l-1]$  by inserting  $y[l]$ :  $E[k, l] = E[k, l-1] + IC(y[l])$
  - extend  $E[k-1, l-1]$  by substituting  $x[k]$  with  $y[l]$ :  
 $E[k, l] = E[k-1, l-1] + SC(x[k], y[l])$

17/56

## Towards a dynamic programming solution (3)

$$E[k, l] = \min( E[k-1, l] + DC(x[k]), \quad // \downarrow \\ E[k, l-1] + IC(y[l]), \quad // \rightarrow \\ E[k-1, l-1] + SC(x[k], y[l]) ) \quad // \swarrow$$

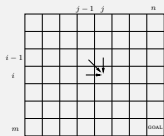
$$E[0, l] = \sum_{i=1}^l IC(y[i]) \\ E[k, 0] = \sum_{i=1}^k DC(x[i])$$

Edit distance =  $E[m, n]$

(Recall:  $m$  and  $n$  are lengths of strings  $x$  and  $y$ )

18/56

## Towards a dynamic programming solution (4)

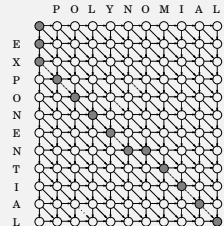


		P	O	L	Y	N	O	M	I	A	L
E	0	1	2	3	4	5	6	7	8	9	10
X	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	7	8	9
A	10	9	8	8	8	7	7	7	7	8	9
L	11	10	9	8	8	8	8	8	8	7	6

$$E[k, l] = \min( E[k-1, l] + DC(x[k]), \quad // \downarrow \\ E[k, l-1] + IC(y[l]), \quad // \rightarrow \\ E[k-1, l-1] + SC(x[k], y[l]) ) \quad // \swarrow$$

19/56

## Towards a dynamic programming solution (5)



$$E[k, l] = \min( E[k-1, l] + DC(x[k]), E[k, l-1] + IC(y[l]), E[k-1, l-1] + SC(x[k], y[l]) )$$

20/56

## Variations

### Approximate prefix:

Is  $y$  approx. prefix of  $x$ ? Decide based on

$$\max_{1 \leq k \leq m} E[k, n]$$

### Approximate suffix:

Initialize  $E[k, 0] = 0$ , use  $E[m, n]$  to determine if  $y$  is an approximate suffix of  $x$

### Approximate substring:

Initialize  $E[k, 0] = 0$ , use  $\max_{1 \leq k \leq m} E[k, n]$  to decide if  $y$  is an approximate substring of  $x$ .

21/56

## More variations

### Supporting transpositions:

Use a fourth term within *min*:

$$E[k-2, l-2] + TC(x[k-1]x[k], y[l-1]y[l])$$

where  $TC$  is a small value for transposed characters, and  $\infty$  otherwise.

22/56

## Similarity Vs Edit-distance

**Edit-distance** cannot be interpreted on its own, and needs to take into account the lengths of strings involved.

**Similarity** can stand on its own.

$$\begin{aligned} S[k, l] &= \max \{ S[k-1, l] - DC(x[k]), & // \downarrow \\ & S[k, l-1] - IC(y[l]), & // \rightarrow \\ & S[k-1, l-1] - SC(x[k], y[l]) \} & // \searrow \end{aligned}$$

$$S[0, l] = - \sum_{i=1}^l IC(y[i])$$

$$S[k, 0] = - \sum_{i=1}^k DC(x[i])$$

- $SC(r, r)$  should be negative, while  $IC$  and  $DC$  should be positive.
- Formulations in biology are usually based on similarity

23/56

## Global alignment (DNA, proteins, ...)

- Similar to edit distance, but uses similarity scores
- Gaps are scored differently: a contiguous sequence of  $n$  deletions does not get penalized as much as  $n$  times a single deletion. (same applies to insertions.)
- Captures the idea that large deletions/insertions are much more likely in nature than many small ones. (Think of chromosomal crossover during meiotic cell division.)
- Obvious formulation to support such gap metrics will lead to more expensive algorithms:  $S[k, l]$  depends on  $S[k-d, l]$  and  $S[k, l-i]$  for any  $d < k$  and  $i < l$ . But a more careful formulation can get back to quadratic time
- Quadratic time still too slow for sequence alignment.

24/56

## Local alignment

- Aimed at identifying local regions of similarity, specifically, the best matches between subsequences of  $x$  and  $y$
- $S[i, j]$  now represents the best alignment of some suffix of  $x[1..j]$  and  $y[1..j]$ .
- A new term is introduced within  $max$ , namely, zero. This means that costs can never become negative.
- In other words, a subsequence does not incur costs because of mismatches preceding the subsequence.
- This change enables regions of similarity to stand out as positive scores.
- Initialize  $F[i, 0] = F[0, j] = 0$

25 / 56

## Improvements to ED Algorithm

**Linear-space:**  $O(mn)$  is not so good for large  $m, n$ .

**Slow** in terms of runtime

**(Possibly) unacceptable** in terms of space usage

- If we are only interested in ED, we can use linear space: retain only the last column computed.
- But if want the actual edits, we need  $O(mn)$  space with the algorithms discussed so far.

Linear-space algorithms developed to overcome this problem.

**Better overall performance:**  $O(md)$  space and runtime if the maximum distance is limited to  $d$ .

In the interest of time, we won't cover these extensions. They are fairly involved, but not necessarily hard.

26 / 56

## LCS application: UNIX diff

Each line is considered a "character:"

- Number of lines far smaller than number of characters
- Difference at the level of lines is easy to convey to users
- Much higher degree of confidence when things line up. Leads to better results on programs.

*But does not work that well on document types where line breaks are not meaningful*, e.g., text files where each paragraph is a line.

Aligns lines that are preserved.

- The edits are then printed in the familiar "diff" format.

27 / 56

## LCS applications: version control, patch,...

Software patches often distributed as "diffs." Programs such as patch can apply these patches to source code or any other file.

**Concurrent updates in version control systems** are resolved using LCS.

- Let  $x$  be the version in the repository
- Suppose that user  $A$  checks it out, edits it to get version  $y$
- Meanwhile,  $B$  also checks out  $x$ , edits it to  $z$ .
- If  $x \rightarrow y$  edits target a disjoint set of locations from those targeted by the  $x \rightarrow z$  edits, both edits can be committed; otherwise a conflict is reported.

28 / 56

## Summary

- A general approach for *optimization problems*
- **Applicable in the presence of:**
  - Optimal substructure
  - A natural ordering among subproblems
  - Numerous subproblems (often, exponential), but only some (polynomial number) are distinct

29 / 56

## Knapsack Problem (Recap)

- You have a choice of items you can pack in the sack
- Maximize value of sack, subject to a weight limit of  $W$

item	calories/lb	weight
bread	1100	5
butter	3300	1
tomato	80	1
cucumber	55	2

**Fractional knapsack:** Fractional quantities acceptable

**0-1 knapsack:** Take all of one item or none at all

**Knapsack w/ repetition:** Take any integral number of items.

No polynomial solution for the last two, but dynamic programming can solve them in **pseudo-polynomial** time of  $O(nW)$ .

30 / 56

## Knapsack w/ repetition: Identify subproblems

- Consider subproblems by reducing the weight
  - Compute  $K(W)$  in terms of  $K(W')$  for  $W' < W$
- Which  $W'$  values to consider?
  - Since the  $i$ th item has a weight  $w_i$ , we should consider only  $W - w_i$  for different  $i$ .
- **Optimal substructure:** If  $K(W)$  is the optimal solution and it includes item  $i$ , then  $K(W) = K(W - w_i) + v_i$

31 / 56

## Knapsack w/ repetition

**KnapWithRep( $w, v, n, W$ )**

```

K[0] = 0
for w = 1 to W do
  K[w] = max1 ≤ i ≤ n, w[i] ≤ w (K[w - w[i]] + v[i])
return K[W]
```

- Fills the array  $K$  from left-to-right
  - If you construct the dag explicitly, you will see that we are looking for the longest path!
- **Runtime:** Outer loop iterates  $W$  times,  $\max$  takes  $O(n)$  time, for a total of  $O(nW)$  time
  - **Not polynomial:** input size logarithmic (not linear) in  $W$ .

32 / 56



## 0-1 Knapsack

Previous algorithm does not work. We need to keep track of which items have been used up.

**Key idea:** Define 2-d array  $K[u, j]$  which computes optimal value for weight  $u$  achievable using items 1.. $j$

- $K[u, j]$  can be computed from  $K[\_, j-1]$ 
  - Either item  $j$  is not included in the optimal solution. Then  $K[u, j] = K[u, j-1]$
  - Or,  $j$  is included, so  $K[u, j] = v[j] + K[u-w[j], j-1]$
- So, fill up the array  $K$  as  $j$  goes from 1 to  $n$
- For each  $j$ , fill  $K$  as  $u$  goes from 1 to  $W$

33 / 56

## 0-1 Knapsack Algorithm

**Knap01**( $w, v, n, W$ )

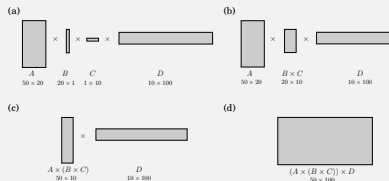
```

 $K[u, 0] = K[0, j] = 0, \forall 1 \leq u \leq W, 1 \leq j \leq n$ 
for  $j = 1$  to  $n$  do
    for  $u = 1$  to  $W$  do
        if  $w[j] > u$  then  $K[u, j] = K[u, j-1]$ 
        else  $K[u, j] = \max(K[u, j-1],$ 
             $K[u-w[j], j-1] + v[j])$ 
    return  $K[W, n]$ 
    
```

**Runtime:** As compared to unbounded knapsack, we have a nested loop here, but the inner loop now executes in  $O(1)$  time. So runtime is still  $O(nW)$

34 / 56

## Chain Matrix Multiplication



	Parenthesization	Cost computation	Cost
	$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120,200
Greedy	$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60,200
	$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

35 / 56

## Chain MM: Formulating Optimal Solution

**Consider outermost multiplication:**

$(M_1 \times \dots \times M_j) \times (M_{j+1} \times \dots \times M_n)$  — we could compute  $j$  using dynamic programming

**Optimal substructure:** Note that the optimal solution for

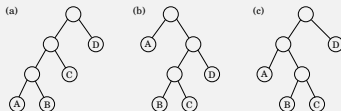
$(M_1 \times \dots \times M_j) \times (M_{j+1} \times \dots \times M_n)$  must rely on optimal solutions to  $M_1 \times \dots \times M_j$  and  $M_{j+1} \times \dots \times M_n$  — or else we could improve the overall solution still

**Cost function:** This suggests a cost function  $C[k, l]$  to denote the optimal cost of  $M_k \times \dots \times M_l$

36 / 56

## Chain MM: Formulating Optimal Solution

**Figure 6.7** (a)  $((A \times B) \times C) \times D$ ; (b)  $A \times ((B \times C) \times D)$ ; (c)  $(A \times (B \times C)) \times D$ .



- Subproblems correspond to one of the subtrees
- Since order of multiplications can't be changed, each subtree must correspond to a "substring" of multiplications, i.e.,  $M_k \times \dots \times M_l$

37 / 56

## Chain MM Algorithm

**chainMM**( $m, n$ )

$C[i, i] = 0 \ \forall 1 \leq i \leq n$

**for**  $s = 1$  to  $n - 1$  **do**

**for**  $k = 1$  to  $n - s$  **do**

$l = k + s$

$C[k, l] = \min_{k \leq i < l} (C[k, i] + C[i + 1, l] + m_{k-1} * m_i * m_l)$

**return**  $C[1, n]$

**Note:**  $m_i$ 's give us the dimension of matrices, specifically,  $M_i$  is an  $m_{i-1} \times m_i$  matrix

**Complexity:**  $O(n^3)$

38 / 56

## Recursive formulation of Dynamic programming

- Recursive formulation can often simplify algorithm presentation, avoiding need for explicit scheduling
- Dependencies between subproblems can be left implicit an equation such as  $K[w] = K[w - w[j]] + v[j]$
- A call to compute  $K[w]$  will automatically result in a call to compute  $K[w - w[j]]$  because of dependency
- *Can avoid solving (some) unneeded subproblems*
- **Memoization:** Remember solutions to function calls so that repeat invocations can use previously returned solutions

39 / 56

## Recursive 0-1 Knapsack Algorithm

**BestVal01**( $u, j$ )

**if**  $u = 0$  **or**  $j = 0$  **return** 0

**if**  $w[j] > u$  **return** **BestVal01**( $u, j - 1$ )

**else return**  $\max(\text{BestVal01}(u, j - 1, v[j]) + \text{BestVal01}(u - w[j], j - 1))$

- Much simpler in structure than iterative version
- Unneeded entries are not computed, e.g.  $\text{BestVal01}(3, \_)$  when all weights involved are even
- **Exercise:** Write a recursive version of ChainMM.

**Note:**  $m_i$ 's give us the dimension of matrices, specifically,  $M_i$  is an  $m_{i-1} \times m_i$  matrix

**Complexity:**  $O(n^3)$

40 / 56

## Dyn. Prog. and Equation Solving

- *The crux of a dynamic programming solution:* set up equation to captures a problem's optimal substructure.  
The equation implies dependencies on subproblem solutions.
- *Dynamic programming algorithm:* finds a schedule that respects these dependencies
- *Typically, dependencies form a DAG:* its topological sort yields the right schedule
- *Cyclic dependencies:* What if dependencies don't form a DAG, but is a general graph.
- *Key Idea:* Use iterative techniques to solve (recursive) equations

41/56

## Fixpoints

- A fixpoint is a solution to an equation:
- Substitute the solution on the rhs, it yields the lhs.

- *Example 1:*  $y = y^2 - 12$ .

- A fixpoint is  $y = 4$ , another is  $y = -3$ .

$$rhs|_{y=4} = 4^2 - 12 = 4 = lhs|_{y=4} \text{ — a fixpoint}$$

42/56

## Fixpoints (2)

- A fixpoint is a solution to an equation:
  - *Example 2:*  $7x = 2y - 4$ ,  $y = x^2 + y/x + 0.5$ .
  - One fixpoint is  $x = 2$ ,  $y = 9$ .

$$rhs_1|_{x=2, y=9} = 14 = lhs_1|_{x=2, y=9}$$

$$rhs_2|_{x=2, y=9} = 9 = lhs_2|_{x=2, y=9}$$

- The term “fixpoint” emphasizes an iterative strategy.
- *Example techniques:* Gauss-Seidel method (linear system of equations), Newton's method (finding roots), ...

43/56

## Convergence

- Convergence is a major concern in iterative methods
  - *For real-values variables*, need to start close enough to the solution, or else the iterative procedure may not converge.
  - *In discrete domains*, rely on *monotonicity* and *well-foundedness*.

**Well-founded order:** An order that has no infinite ascending chain (i.e., sequence of elements  $a_0 < a_1 < a_2 < \dots$  where there is no maximum)

**Monotonicity:** Successive iterations produce larger values with respect to the order, i.e.,  $rhs|_{sol^i} \geq sol^i$

**Result:** Start with an initial guess  $S^0$ , note  $S^i = rhs|_{S^{i-1}}$ .

- Due to monotonicity,  $S^i \geq S^{i-1}$ , and
- by well-foundedness, the chain  $S^0, S^1, \dots$  can't go on forever.
- Hence iteration must converge, i.e.,  $\exists k \forall i > k \ S^i = S^k$

44/56

## Role of Iterative Solutions

- **Fixpoint iteration resembles an inductive construction**
  - $S^0$  is the base case,  $S^i$  construction from  $S^{i-1}$  is the induction step.
- Drawback of **explicit fixpoint iteration**: hard to analyze the number of iterations, and hence the runtime complexity
- So, algorithms tend to rely on inductive, bottom-up constructions with enough detail to reason about runtime.
- Fixpoint iteration thus serves two main purposes:
  - When it is possible to bound its complexity in advance, e.g., non-recursive definitions
  - As an intermediate step that can be manually analyzed to uncover inductive structure explicitly.

45/56

## Shortest Path Problems

**Graphs with cycles:** Natural example where the optimal substructure equations are recursive.

**Single source:**  $d_v = \min_{u|(u,v) \in E} (d_u + l_{uv})$

**All pairs:**  $d_{uv} = \min_{w|(w,v) \in E} (d_{uw} + l_{wv})$

or, alternatively,  $d_{uv} = \min_{w \in V} (d_{uw} + d_{wv})$

*Our study of shortest path algorithms is based on fixpoint formulation*

- Shows how different shortest path algorithms can be derived from this perspective
- Highlights the similarities between these algorithms, making them easier to understand/remember

46/56

## Single-source shortest paths

For the source vertex  $s$ ,  $d_s = 0$ . For  $v \neq s$ , we have the following equation that captures the optimal substructure of the problem. We use the convention  $l_{uv} = 0$  for all  $u$ , as it simplifies the equation:

$$d_v = \min_{u|(u,v) \in E} (d_u + l_{uv})$$

Expressing edge lengths as a matrix, this equation becomes:

$$\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} l_{12} & l_{13} & \cdots & l_{1n} \\ l_{22} & l_{23} & \cdots & l_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ l_{j2} & l_{j3} & \cdots & l_{jn} \\ \vdots & \vdots & \ddots & \vdots \\ l_{n2} & l_{n3} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_j \\ \vdots \\ d_n \end{bmatrix}$$

Matches the form of linear simultaneous equations, except that point-wise multiplication and addition become the integer “+” and  $\min$  operations respectively.

47/56

## Single-source shortest paths

SSP, written as a recursive matrix equation is:

$$D = \mathbf{L}D$$

Now, solve this equation iteratively:

$$D^0 = Z \quad (Z \text{ is the column matrix consisting of all } \infty \text{ except } d_s = 0)$$

$$D^1 = \mathbf{L}Z$$

$$D^2 = \mathbf{L}D^1 = \mathbf{L}(\mathbf{L}Z) = \mathbf{L}^2Z$$

Or, more generally,  $D^j = \mathbf{L}^jZ$

- $\mathbf{L}$  is the generalized adjacency matrix, with entries being edge weights (aka edge lengths) rather than booleans.
- Side note: In this domain, multiplicative identity  $\mathbf{I}$  is a matrix with zeroes on the main diagonal, and  $\infty$  in all other places.
  - So,  $\mathbf{L} = \mathbf{I} + \mathbf{L}$ , and hence  $\mathbf{L}^* = \lim_{j \rightarrow \infty} \mathbf{L}^j$

48/56

## Single-source shortest paths

- Recall the connection between paths and the entries in  $\mathbf{L}$ .
- Thus,  $D^i$  represents the shortest path using  $i$  or fewer edges!
- Unless there are cycles with negative cost in the graph, all shortest paths must have a length less than  $n$ , so:

- $D^n$  contains all of the shortest paths from the source vertex  $s$
- $d_i^n$  is the shortest path length from  $s$  to the vertex  $i$ .

Computing  $\mathbf{L} \times \mathbf{L}$  takes  $O(n^3)$ , so overall SSP cost is  $O(n^4)$ .

49 / 56

## SSP: Improving Efficiency of Matrix Formulation

- Compute the product from right:  $(\mathbf{L} \times (\mathbf{L} \times \cdots (\mathbf{L} \times \mathbf{Z}) \cdots))$ 
  - Each multiplication involves  $n \times n$  and  $1 \times n$  matrix, so takes  $O(n^2)$  instead of  $O(n^3)$  time.
  - Overall time reduced to  $O(n^3)$ .
- To compute  $\mathbf{L} \times d_j$ , enough to consider neighbors of  $j$ , and not all  $n$  vertices

$$d_j^i = \min_{k|(k,j) \in E} (d_k^{i-1} + l_{kj})$$

- Computes each matrix multiplication in  $O(|E|)$  time, so we have an overall  $O(|E||V|)$  algorithm.
- We have stumbled onto the Bellman-Ford algorithm!*

50 / 56

## Further Optimization on Iteration

$$d_j^i = \min_{k|(k,j) \in E} (d_k^{i-1} + l_{kj})$$

- Optimization 1:** If none of the  $d_k$ 's on the rhs changed in the previous iteration, then  $d_j^i$  will be the same as  $d_j^{i-1}$ , so we can skip recomputing it in this iteration.
- Can be an useful improvement in practice, but asymptotic complexity unchanged from  $O(|V||E|)$

51 / 56

## Optimizing Iteration

$$d_j^i = \min_{k|(k,j) \in E} (d_k^{i-1} + l_{kj})$$

**Optimization 2:** Wait to update  $d_j$  on account of  $d_k$  on the rhs *until*  $d_k$ 's cost stabilizes

- Avoids repeated propagation of min cost from  $k$  to  $j$  — instead propagation takes place just once per edge, i.e.,  $O(|E|)$  times
- If all weights are non-negative, we can determine when costs have stabilized for a vertex  $k$ 
  - There must be at least  $r$  vertices whose shortest path from the source  $s$  uses  $r$  or fewer edges.
  - In other words, if  $d_k^i$  has the  $r$ th lowest value, then  $d_k^i$  has stabilized if  $r \leq i$

**Voila!** We have Dijkstra's Algorithm!

52 / 56

## All pairs Shortest Path (I)

$$d_{uv}^i = \min_{w|(w,v) \in E} (d_{uw}^{i-1} + l_{wv})$$

- Note that  $d_{uv}$  depends on  $d_{uw}$ , but not on any  $d_{xy}$ , where  $x \neq u$ .
- So, solutions for  $d_{xy}$  don't affect  $d_{uv}$ .
- i.e., we can solve a separate SSP, each with one of the vertices as source
- i.e., we run Dijkstra's  $|V|$  times, overall complexity  $O(|E||V| \log |V|)$

53/56

## All pairs Shortest Path (II)

$$d_{uv}^i = \min_{w \in E} (d_{uw}^{i-1} + d_{wv}^{i-1})$$

Matrix formulation:

$$\mathbf{D} = \mathbf{D} \times \mathbf{D}$$

with  $\mathbf{D}^0 = \mathbf{L}$ .

Iterative formulation of the above equation yields

$$\mathbf{D}^i = \mathbf{L}^{2^i}$$

We need only consider paths of length  $\leq n$ , so stop at  $i = \log n$ . Thus, overall complexity is  $O(n^3 \log n)$ , as each step requires  $O(n^3)$  multiplication.

*We have just uncovered a variant of Floyd-Warshall algorithm!*

- Typically used with matrix-multiplication based formulation.
- Matches ASP I complexity for dense graphs ( $|E| = \Theta(|V|^2)$ )*

54/56

## Further Improving ASP II

Each step has  $O(n^3)$  complexity as it considers all  $(u, w, v)$  combinations

*Note:* Blind fixpoint iteration “breaks” recursion by limiting path length.

- Converts  $d_{uv}$  into  $d_{uv}^i$  where  $i$  is the path length
- Worked well for SSP & ASP I, not so well for ASP II

*Can we break cycles by limiting something else*, say, vertices on the path?

*Floyd-Warshall:* Define  $d_{uv}^k$  as the shortest path from  $u$  to  $v$  that only uses intermediate vertices 1 to  $k$ .

$$d_{uv}^k = \min(d_{uv}^{k-1}, d_{uk}^{k-1} + d_{kv}^{k-1})$$

*Complexity:* Need  $n$  iterations to consider  $k = 1, \dots, n$  but each iteration considers only  $n^2$  pairs, so overall runtime becomes  $O(n^3)$

55/56

## Summary

- A versatile, robust technique to solve optimization problems
- *Key step:* Identify *optimal substructure* in the form of an equation for optimal cost
- If equations are non-recursive, then either
  - identify underlying DAG, compute costs in topological order, or,
  - write down a memoized recursive procedure
- For recursive equations, “break” recursion by introducing additional parameters.
  - A fixpoint iteration can help expose such parameters.
- Remember the choices made while computing the optimal cost, use these to construct optimal solution.

56/56