CS583 Lecture 10

Jana Kosecka

Graph Algorithms
Shortest Path Algorithms
Dynamic Programming

Many slides here are based on D. Luebke slides

Previously

- Depth first search
- DAG's
  - topological sort
  - strongly connected components
- MST
  - Prim's
  - Kruskal's
- Shortest path with non-negative weights

Shortest Path Algorithms

Single-Source Shortest Path

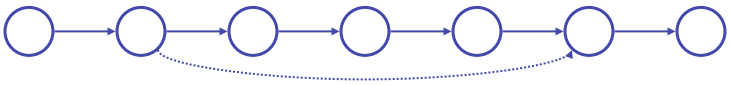
- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v

- "Shortest-path" = minimum weight
- Weight of path is sum of edges

- E.g., a road map: what is the shortest path from Faixfax to Washington DC?

Shortest Path Properties

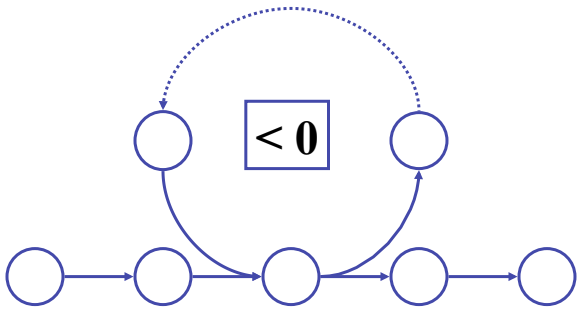
- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



- Proof: suppose some subpath is not a shortest path
  There must then exist a shorter subpath
  Could substitute the shorter subpath for a shorter path
  but then overall path is not shortest path. Contradiction
- *Optimal substructure property* – hallmark of dynamic programming

Shortest Path Properties

- In graphs with negative weight cycles, some shortest paths will not exist (*Why?*):

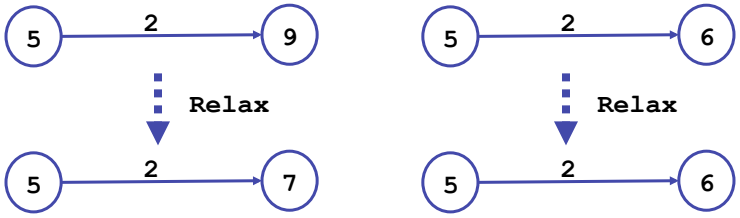


Relaxation

- A key technique in shortest path algorithms is *relaxation*
  Idea: for all *v*, maintain upper bound d[*v*] on δ(s,v)

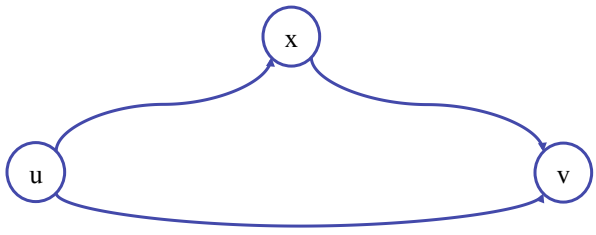
```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```

- **Relaxing an edge – checking if it can improve**
- **The cost of the path**



Shortest Path Properties

- Define δ(u,v) to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: δ(u,v) ≤ δ(u,x) + δ(x,v)
- "Proof":



This path is no longer than any other path

Shortest Path Properties

- Triangle inequality
- Upper bound property
- No path property
- Convergence Property
- Path relaxation property
- Predecessor subgraph property

Dijkstra's Algorithm

- If no negative edge weights, we can beat Bellman-Ford
- Similar to breadth-first search
- Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  Use a priority queue keyed on d[v]

Bellman Ford Algorithm

- Single source shortest path algorithm
- Weights can be negative

- Algorithm returns NIL if there is negative weight cycle
- Otherwise returns produces shortest paths from source
- to all other vertices

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

Initialize d[], which will converge to shortest-path value δ

Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution
Under what condition do we get a solution?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

What will be the running time?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```

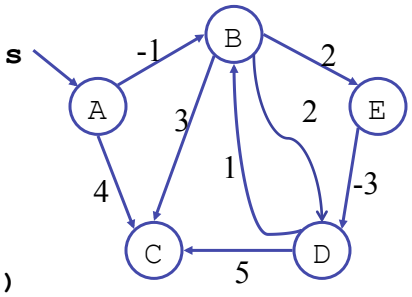
What will be the running time?

A: O(VE)

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford Algorithm

```
BellmanFord()
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0;
   for i=1 to |V|-1
      for each edge (u,v) ∈ E
         Relax(u,v, w(u,v));
   for each edge (u,v) ∈ E
      if (d[v] > d[u] + w(u,v))
           return "no solution";
```



Ex: work on board

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

Bellman-Ford

- Running time O(VE)
- Not so good for large dense graphs
- Still very practical

Bellman-Ford

- Prove: Algorithm after |V|-1 passes of the **for** loop, all *d* values (shortest path values) are correct (for all vertices)
- Consider shortest path from s to v, since the path is simple (no loops) there are at most |V|-1 edges
- $s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v$

Initially, d[s] = 0 is correct, and doesn't change (*Why?*)
After 1 pass through edges, $d[v_1]$ is correct (*Why?*) and doesn't change
After 2 passes, $d[v_2]$ is correct and doesn't change
…
Terminates in |V| - 1 passes: (*Why?*)
What if it doesn't?

Bellman-Ford

- Note that order in which edges are processed affects how quickly it converges
- Correctness: show d[v] = δ(s,v) for all vertices or returns FALSE (negative weight cycle)

By previous lemma at the termination we have
d[v] = δ(s,v)
<= δ(s,u) + w(u,v) by triangle inequality
= d[u] + w(u,v)
And none of the test in the last loop will return "no solution", i.e. TRUE.

```
if (d[v] > d[u] + w(u,v))
    return "no solution";
```

If there is a negative weight cycle – prove by contradiction that Bellman Ford algorithm will return FALSE (see book)

DAG Shortest Paths

- Problem: finding shortest paths in DAG
  Bellman-Ford takes O(VE) time.
  How can we do better?

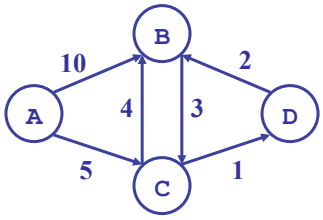
  - Idea: use topological sort
  - If were lucky and process vertices on each shortest path from left to right, would be done in one pass
  - Every path in a DAG is subsequence of topologically sorted vertex order, so processing vertices in that order, we will do each path in forward order (will never relax edges out of vertex before doing all edges into vertex).
  - Thus: just one pass. *What will be the running time?*

Review: Dijkstra's Algorithm

```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

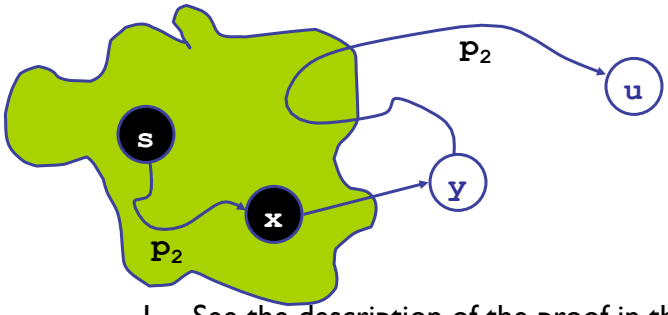
Relaxation Step

Note: this is really a call to Q->DecreaseKey()



Ex: run the algorithm

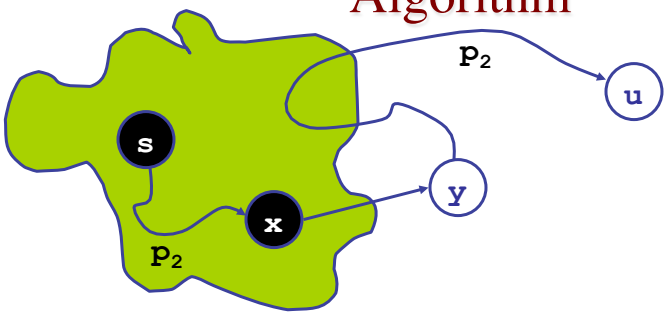
Correctness Of Dijkstra's Algorithm



1. See the description of the proof in the book

Show that Dijkstra's algorithm will terminate with
The cost of each node to be the cost of shortest path.
Idea: show that when the vertex is added to the set the cost of that vertex is the length of the shortest path
Reminder: We always add the vertex with minimal cost

Correctness Of Dijkstra's Algorithm



- Want to show that when vertex is added to set S, $d[u] = \delta(s,u)$
- and throughout note that $d[u] \geq \delta(s,u)$ $\forall u$
- Proof by contradiction $d[u]$ is not equal to $\delta(s,u)$
- Before u gets added, some other vertex y on that shortest path needs to be added; claim that $d[y] = \delta(s,y)$ when added.
- Know that $d[x] = \delta(s,x)$ and $\delta(s,y) <= \delta(s,u)$ and $d[y] = \delta(s,y)$, so $d[y] <= d[u]$
- But both y and u are outside of S when is chosen so $d[u] <= d[y]$
- Hence $d[y] = d[u] = \delta(s,y) = \delta(s,y)$

Previously

- Shortest path algorithms: given single source compute shortest path to all other vertices.
- Dijkstra *O(E + V lg V)*
- Bellman-Ford *O(VE)*
- DAG *O(V+E)*
- All pairs shortest path: compute shortest path between each pair of nodes
- Option 1. Run single source shortest path from each node using previous algorithms
- Run Bellman-Ford once for each vertex $O(V^2E)$
- Can we do better ? See All-pairs-shortest path

Dynamic Programming

Chap 15.

Dynamic Programming

- Another strategy for designing algorithms is *dynamic programming*
- A metatechnique, not an algorithm (like divide & conquer)
- The word "programming" is historical and predates computer programming
- Use when problem breaks down into recurring small subproblems

Dynamic Programming History

• Bellman. Pioneered the systematic study of dynamic programming in the 1950s.

• Etymology.
Dynamic programming = planning over time.
Secretary of Defense was hostile to mathematical research.
Bellman sought an impressive name to avoid confrontation.
"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Reference: Bellman, R. E. *Eye of the Hurricane, An Autobiography.*

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….
- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Dynamic Programming

- More Examples

- Matrix chain multiplication
- Longest common subsequence
- Optimal triangulation
- All-pairs-shortest path

Dynamic Programming

- Problem solving methodology (as divide and conquer)
- Idea: divide into sub-problems, solve sub-problems
- Applicable to optimization problems
- Ingredients

1. Characterize the optimal solution
2. Recursively define a value of the optimal solution
3. Compute values of optimal solution bottom up
4. Construct an optimal solution from computed inf.

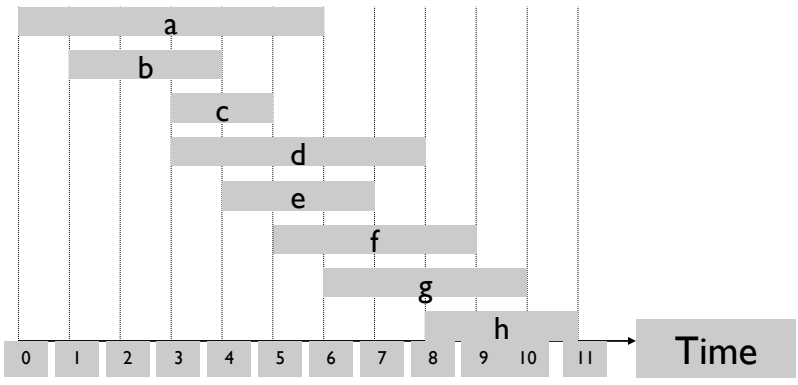
Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to sub-problems (*optimal substructure*)
- Algorithm finds solutions to sub-problems and stores them in memory for later use
- More efficient than "*brute-force methods*", which solve the same sub-problems over and over again

Weighted Interval Scheduling

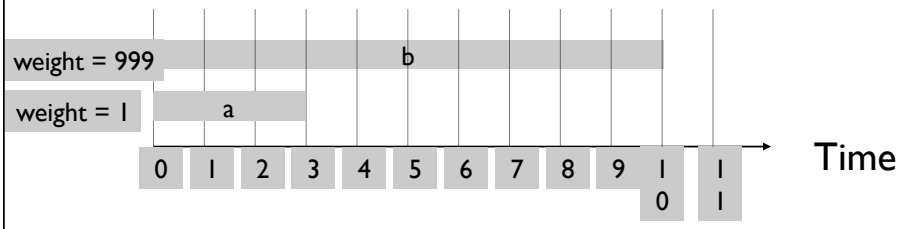
Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



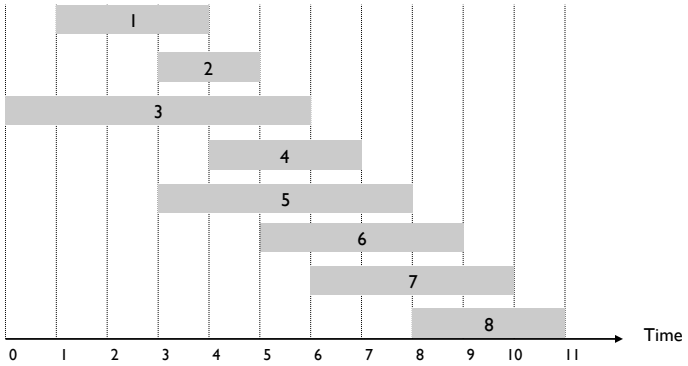
Unweighted Interval Scheduling Review

- Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def. p(j) = largest index i < j such that job i is compatible with j. Ex: p(8) = 5, p(7) = 3, p(2) = 0.



Dynamic Programming: Binary Choice

• Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

optimal substructure

Case 1: OPT selects job j.

can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

Case 2: OPT does not select job j.

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

Weighted Interval Scheduling: Brute Force

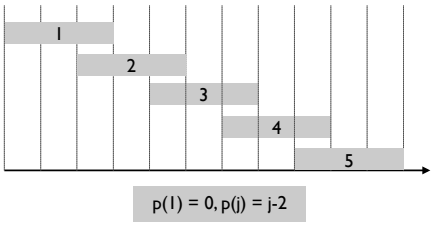
• Brute force recursive implemenation

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤
fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-
Opt(j-1))
}
```

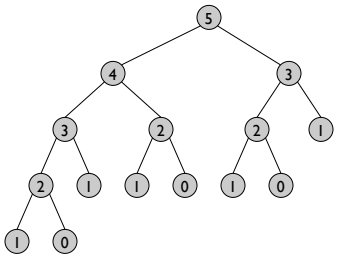
Weighted Interval Scheduling: Brute Force

•Observation. Recursive algorithm fails spectacularly because of redundant sub-problems ⇒ exponential algorithms.

•Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



Weighted Interval Scheduling: Memoization

•Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n ← global array
   M[j] = empty
M[j] = 0

M-Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(wj + M-Compute-Opt(p(j)), M-Compute-
Opt(j-1))
   return M[j]
}
```

Weighted Interval Scheduling: Running Time

•Claim. Memoized version of algorithm takes O(n log n) time.
Sort by finish time: O(n log n).
Computing p(·): O(n) after sorting by start time.
`M-Compute-Opt(j)`: each invocation takes O(1) time and either
(i) returns an existing value `M[j]`
(ii) fills in one new entry `M[j]` and makes two recursive calls
Progress measure Φ = # nonempty entries of `M[]`.
initially $\Phi = 0$, throughout $\Phi \leq n$.
(ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.
Overall running time of `M-Compute-Opt(n)` is O(n). ▪

•Remark. O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

•Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
•A. Do some post-processing.

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

Weighted Interval Scheduling: Bottom-Up

•Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

Dynamic Programming Example: Longest Common Subsequence

- *Longest common subsequence* (*LCS*) problem:
  Given two sequences x[1..m] and y[1..n], find the longest subsequence which occurs in both
  Ex: x = {A B C B D A B }, y = {B D C A B A}
  {B C} and {A A} are both subsequences of both
  What is the LCS?
  Brute-force algorithm: For every subsequence of x, check if it's a subsequence of y

  How many subsequences of x are there?
  What will be the running time of the brute-force alg?

Longest Common Subsequence (LCS)

- Application: comparison of two DNA strings
- Ex: X= {A B C B D A B }, Y= {B D C A B A}
- Longest Common Subsequence:
- X = A **B** **C** **B** D **A** B
- Y = **B** D **C** A **B** **A**
- Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- Brute-force algorithm: 2^m subsequences of x to check against n elements of y: $O(n\ 2^m)$
- We can do better: for now, let's only worry about the problem of finding the *length* of LCS
- When finished we will see how to backtrack from this solution back to the actual LCS
- Notice LCS problem has optimal substructure
  Subproblems: LCS of pairs of *prefixes* of x and y

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define *c[i,j]* to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be *c[m,n]*

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. *c[0,0] = 0*)
- LCS of empty string and any other string is empty, so for every i and j: *c[0, j] = c[i,0] = 0*

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j],* we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- **Second case:** *x[i] != y[j]*
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1},Y_{j-1}) ?

LCS Length Algorithm

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m        c[i,0] = 0       // special case: Y0
4. for j = 1 to n        c[0,j] = 0       // special case: X0
5. for i = 1 to m                         // for all Xi
6.       for j = 1 to n                   // for all Yj
7.             if ( Xi == Yj )
8.                   c[i,j] = c[i-1,j-1] + 1
9.             else c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X,Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

ABCB
BDCAB

LCS Example (0)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

ABCB
BDCAB

LCS Example (1)

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

ABCB
BDCAB

LCS Example (2)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0				
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (3)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0		
2	B		0					
3	C		0					
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (4)

i	Xi	j=0, Yj	1, B	2, D	3, C	4, A	5, B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (5)

i	Xi	j=0, Yj	1, B	2, D	3, C	4, A	5, B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1 →	1
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (6)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (7)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (8)

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (10)

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1			
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (11)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (12)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0					

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (13)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (14)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	

if ($X_i == Y_j$)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

ABCB
BDCAB

LCS Example (15)

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

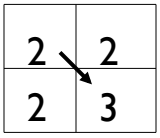
O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]*
or *c[i-1, j-1]*
For each c[i,j] we can say how it was acquired:



For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

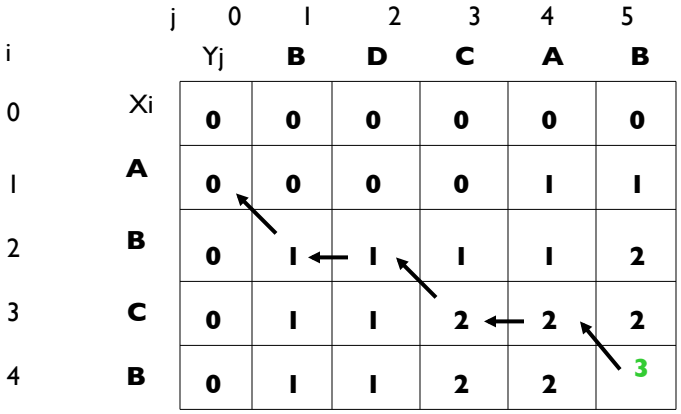
How to find actual LCS - continued

- Remember that

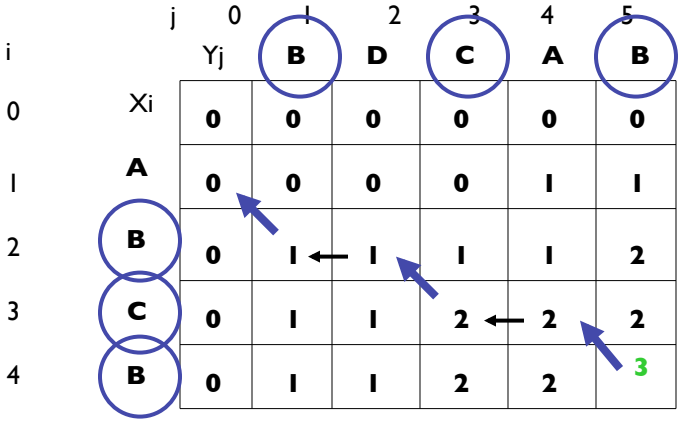
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]*
- (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS



Finding LCS (2)



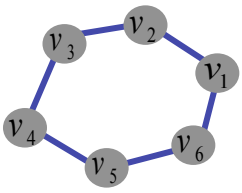
LCS (reversed order): B C B

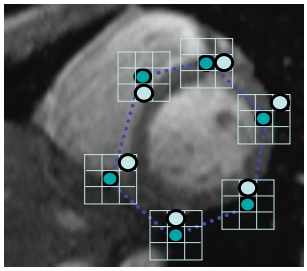
LCS (straight order):

Review: Dynamic Programming

- Summary of the basic idea:
- Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
- Overlapping subproblems: few subproblems in total, many recurring instances of each
- Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Variations:
  "Table" could be 3-dimensional, triangular, a tree, etc.

Computer Vision: Energy minimization: dynamic programming





With this form of the energy function, we can minimize using dynamic programming, with the *Viterbi* algorithm.

Iterate until optimal position for each point is the center of the box, i.e., the snake is optimal in the local search space constrained by boxes.

Fig from Y. Boykov
[Amini, Weymouth, Jain, 1990]

Energy minimization: dynamic programming

- Possible because snake energy can be rewritten as a sum of pair-wise interaction potentials:

$$E_{total}(v_1,\ldots,v_n) = \sum_{i=1}^{n-1} E_i(v_i, v_{i+1})$$

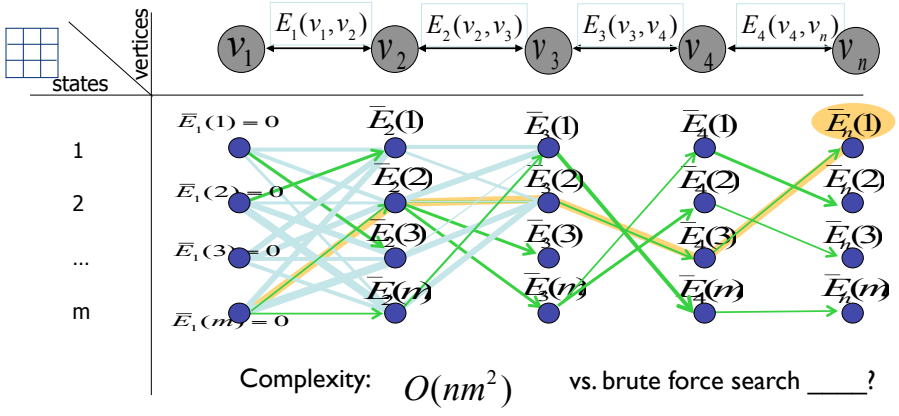
  - Or sum of triple-interaction potentials.

$$E_{total}(v_1,\ldots,v_n) = \sum_{i=1}^{n-1} E_i(v_{i-1}, v_i, v_{i+1})$$

Viterbi algorithm

Main idea: determine optimal position (state) of predecessor, for each possible position of self. Then backtrack from best state for last vertex.

$$E_{total} = E_1(v_1,v_2) + E_2(v_2,v_3) + \ldots + E_{n-1}(v_{n-1},v_n)$$



Complexity: $O(nm^2)$ vs. brute force search ____?

Example adapted from Y. Boykov

Viterbi alg. dynamic programming

- We are interested in the assignment of states for vertices, $v_1, v_2, \ldots v_n$ such that total energy is minimized. Each vertex can be in one of the *m* states $s_1, s_2, \ldots s_m$

$$\min_{v_n} \min_{v_{n-1}} \ldots \min_{v_1} = [E_1(v_1, v_2) + E_2(v_2, v_3) + \ldots + E_{n-1}(v_{n-1}, v_n)]$$

$$\min_{v_n} \min_{v_{n-1}} = [E_{n-1}(v_{n-1}, v_n) + \ldots + E_1(v_2, v_3) + \min_{v_1} E_2(v_1, v_2)]$$

$$\min_{v_n} \min_{v_{n-1}} = [E_{n-1}(v_{n-1}, v_n) + \ldots + E_1(v_2, v_3) + M_{1,2}(v_1)]$$

.

.

memoize the partial solution

$$\min_{v_n} = [E_{n-1}(v_{n-1}, v_n) + M_{n-2,n-1}(x_{n-1})]$$

Matrix Chain Multiplication

- Given sequence of matrices $A_1 A_2 \cdots A_n$
- And their dimensions $p_0, p_1, p_2, \cdots p_n$
- What is the optimal order of multiplication
- Example: why two different orders matter ?

- Brute force strategy – examine all possible parenthezations

$$P(n) = \sum_{k=1}^{n-1} P(k) P(n-k)$$

- Solution to the recurrence

$$P(n) = \Omega(4^n / n^{3/2})$$

Matrix Chain Multiplication

- Substructure property

$$A_1 A_2 \cdots A_k A_{k+1} \cdots A_n$$

- Total cost will be cost of solving the two subproblems and multiplying the two resulting matrices

$$(A_1 A_2 \cdots A_k)(A_{k+1} \cdots A_n)$$

- Optimal substructure find the split which will yield the minimal total cost
- Idea: try to define it recursively

Matrix Chain Multiplication

- Define the cost recursively m[i,j] cost of multiplying

$$A_i \cdots A_j$$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{otherwise} \end{cases}$$

Matrix Chain Multiplication

- Option 1: Compute the cost recursively, remember good splits
- Draw the recurrence tree for $A_1 \cdots A_4$

Matrix Chain Multiplication

- Look up the pseudo-code in the textbook
- Core is the recursive call

C = RecursiveMatrixChain(p,i,k) +
RecursiveMatrixChain(p,k+1,j) + $p_{i-1}p_k p_j$

$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1)$$

$$T(n) \geq 1 + 2\sum_{i=1}^{n-1} T(i) + n$$

- Prove by substitution

$$T(n) = \Omega(2^n)$$

- Recursive solution would still take exponential time ☹

Matrix Chain Multiplication

- Idea: memoization
- Look at the recursion tree, many of the sub-problems repeat
- Remember then and reuse in the
- How many sub-problems do we have ? $\binom{n}{2} + n$
- Why ?

$$T(n) = \Theta(n^2)$$

- Compute the solution to all subproblems bottom up
- Memoize in the table store intermediate cost *m[i,j]*

Matrix Chain Multiplication

```
1. n = length(p)-1
2. for i=1 to n m[i,i] = 0;  % initialize
3. for l=2 to n                          %  l is the chain length
4.    for i=1 to n-l+1              % first compute all m[i,i
   +1], then m[i,i+2]
5.           do j := i+1-1
6.              m[i,j] ← inf
7.              for k = i to j-1
8.                     do q = m[i,k] + m[k+1,j] +
   p(i-1)p(k)p(j)
9.                     if q < m[i,j] then
10.                       m[i,j] = q;
11.                        s[i,j] = k; % remember k with min
   cost
12.                    end
13.             end
14.    end
15.Return m and s
```

Matrix Chain Multiplication

- Example

Dynamic Programming

- What is the structure of the sub-problem
- Common pattern:
- Optimal solution requires making a choice which leads to optimal solution
- Hard part: what is the optimal subproblem structure
  How many subproblems ?
  How many choices we have which sub-problem to use ?

- Matrix chain multiplication

- LCS

Dynamic Programming

- What is the structure of the sub-problem
- Common pattern:
- Optimal solution requires making a choice which leads to optimal solution
- Hard part: what is the optimal subproblem structure
  How many sub-problems ?
  How many choices we have which sub-problem to use ?

- Matrix chain multiplication: 2 subproblems, j-i choices

- LCS: 3 suproblems 3 choices

- Subtleties (graph examples) shortest path, longest path

Previously

- Shortest path algorithms: given single source compute shortest path to all other vertices.
- Dijkstra $O(E + V \lg V)$
- Bellman-Ford $O(VE)$
- DAG $O(V+E)$
- All pairs shortest path: compute shortest path between each pair of nodes
- Option 1. Run single source shortest path from each node using previous algorithms
- Run Bellman-Ford once for each vertex $O(V^2E)$
- Can we do better ?

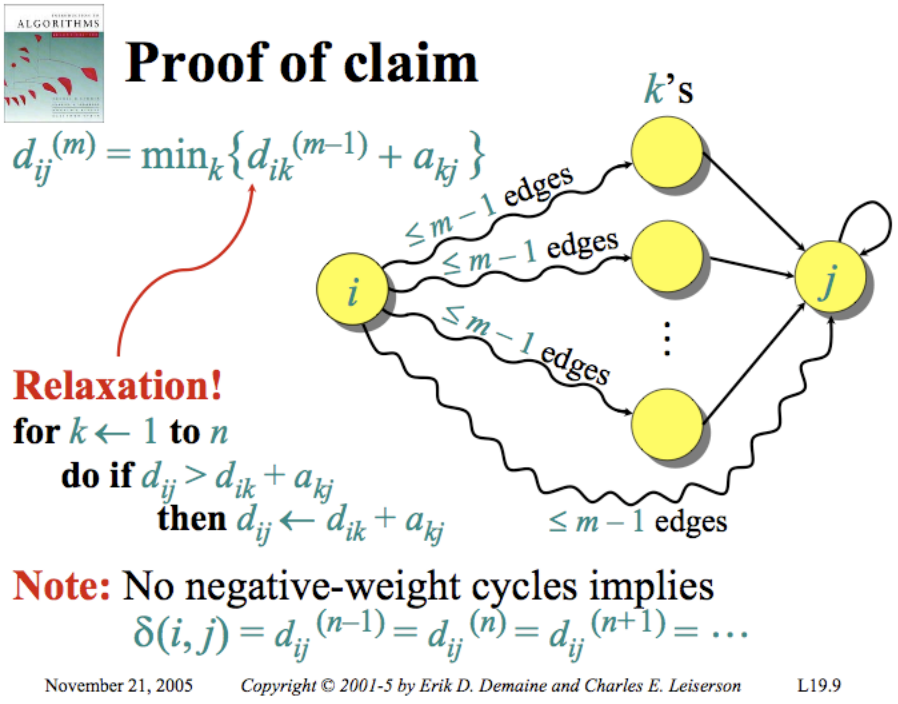
All pairs shortest path

- Final representation of the solution is in adjacency matrix
- δ(i,j) will be the length of the shortest path from *i* to *j*
- Structure of the optimal solution

$$d_{ij}^{0} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

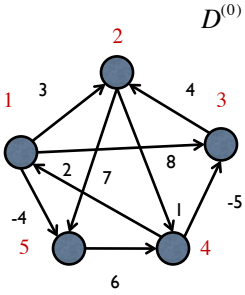
$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\})$$

- Weight of the shortest path with *m-1* edges and minimum of the weight of any path consisting of at most *m* edges



Example all shortest paths

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$



$D^{(0)} = W$

$D^{(1)} = D^{(0)}W = WW$

$D^{(2)} = D^{(1)}W = D^{(0)}WW$

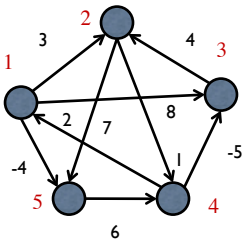
- Matrix multiplication $\Theta(n^4)$
- Repeated Squaring $\Theta(n^3 \lg n)$

$$d_{ij}^{(m)} = \min(d_{ij}^{(m-1)}, \min_{1 \le k \le n}\{d_{ik}^{(m-1)} + w_{kj}\})$$

Like matrix multiplication + => min . => +

Example all shortest paths

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(3)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$



$D^{(0)} = W$

$D^{(1)} = D^{(0)}W = WW$

$D^{(2)} = D^{(1)}W = D^{(0)}WW$

- Matrix multiplication $\Theta(n^4)$
- Repeated Squaring $\Theta(n^3 \lg n)$