CS38
Introduction to Algorithms

Lecture 10

May 1, 2014

Outline

- Dynamic programming design paradigm
  - longest common subsequence
  - edit distance/string alignment
  - shortest paths revisited: Bellman-Ford
  - detecting negative cycles in a graph
  - all-pairs-shortest paths

* some slides from Kevin Wayne

Dynamic programming

"programming" = "planning"

"dynamic" = "over time"

- basic idea:
  - identify subproblems
  - express solution to subproblem in terms of other "smaller" subproblems
  - build solution bottom-up by filling in a table
- defining subproblem is the hardest part

Dynamic programming summary

- identify subproblems:
  - present in recursive formulation, or
  - reason about what residual problem needs to be solved after a simple choice
- find order to fill in table
- running time (size of table)·(time for 1 cell)
- optimize space by keeping partial table
- store extra info to reconstruct solution

Longest common subsequence

- Two strings:
  - $x = x_1\ x_2\ \ldots\ x_m$
  - $y = y_1\ y_2\ \ldots\ y_n$
- Goal: find longest string z that occurs as subsequence of both.

  e.g. x = gctatcgatctagcttata

  y = catgcaagcttgcactgatctcaaa

  z = tattctcta

Longest common subsequence

- Two strings:
  - $x = x_1\ x_2\ \ldots\ x_m$
  - $y = y_1\ y_2\ \ldots\ y_n$
- Goal: find longest string z that occurs as subsequence of both.

  e.g. x = gctatcgatctagcttata

  y = catgcaagcttgcactgatctcaaa

  z = tattctcta

Longest common subsequence

- Two strings:
  - $x = x_1\ x_2 \ldots x_m$
  - $y = y_1\ y_2 \ldots y_n$
- structure of LCS: let $z_1\ z_2 \ldots z_k$ be LCS of $x_1\ x_2 \ldots x_m$ and $y_1\ y_2 \ldots y_n$
  - if $x_m = y_n$ then $z_k = x_m = y_n$ and $z_1\ z_2 \ldots z_{k-1}$ is LCS of $x_1\ x_2 \ldots x_{m-1}$ and $y_1\ y_2 \ldots y_{n-1}$

Longest common subsequence

- Two strings:
  - $x = x_1\ x_2 \ldots x_m$
  - $y = y_1\ y_2 \ldots y_n$
- structure of LCS: let $z_1\ z_2 \ldots z_k$ be LCS of $x_1\ x_2 \ldots x_m$ and $y_1\ y_2 \ldots y_n$
  - if $x_m \neq y_n$ then
    - $z_k \neq x_m \Rightarrow z$ is LCS of $x_1\ x_2 \ldots x_{m-1}$ and $y_1\ y_2 \ldots y_n$
    - $z_k \neq y_n \Rightarrow z$ is LCS of $x_1\ x_2 \ldots x_m$ and $y_1\ y_2 \ldots y_{n-1}$

Longest common subsequence

- Two strings:
  - $x = x_1\ x_2 \ldots x_m$
  - $y = y_1\ y_2 \ldots y_n$
- Subproblems: prefix of x, prefix of y

  OPT(i,j) = length of LCS for $x_1\ x_2 \ldots x_i$ and $y_1\ y_2 \ldots y_j$
- using structure of LCS: OPT(i,j) =

0	if i = 0 or j = 0
OPT(i-1,j-1) + 1	if $x_i = y_j$
max{OPT(i,j-1),OPT(i-1, j)}	if $x_i \neq y_j$

Longest common subsequence

- what order to fill in the table?

```
LCS-length(x, y: strings)
OPT(i, 0)  = 0 for all i
OPT(0, j)  = 0 for all j
for i = 1 to m
  for j = 1 to n
    if x_i = y_j then OPT(i,j) = OPT(i-1, j-1) + 1
    elseif OPT(i-1, j) ≥ OPT(i,j-1) then OPT(i,j) = OPT(i-1, j)
    else OPT(i,j) = OPT(i,j-1)
return(OPT(n,m))
```


Longest common subsequence

```
LCS-length(x, y: strings)
OPT(i, 0)  = 0 for all i
OPT(0, j)  = 0 for all j
for i = 1 to m
  for j = 1 to n
    if x_i = y_j then OPT(i,j) = OPT(i-1, j-1) + 1
    elseif OPT(i-1, j) ≥ OPT(i,j-1) then OPT(i,j) = OPT(i-1, j)
    else OPT(i,j) = OPT(i,j-1)
return(OPT(n,m))
```

- running time?
  - O(mn)

Longest common subsequence

```
LCS-length(x, y: strings)
OPT(i, 0)  = 0 for all i
OPT(0, j)  = 0 for all j
for i = 1 to m
  for j = 1 to n
    if x_i = y_j then OPT(i,j) = OPT(i-1, j-1) + 1
    elseif OPT(i-1, j) ≥ OPT(i,j-1) then OPT(i,j) = OPT(i-1, j)
    else OPT(i,j) = OPT(i,j-1)
return(OPT(n,m))
```

- space O(nm)
  - can be improved to O(min{n,m})

Longest common subsequence

```
LCS-length(x, y: strings)
OPT(i, 0)  = 0 for all i
OPT(0, j)  = 0 for all j
for i = 1 to m
  for j = 1 to n
     if x_i = y_j then OPT(i,j) = OPT(i-1, j-1) + 1
     elseif OPT(i-1, j) ≥ OPT(i,j-1) then OPT(i,j) = OPT(i-1, j)
     else OPT(i,j) = OPT(i,j-1)
return(OPT(n,m))
```

- reconstruct LCS?
  - store which of 3 cases was taken in each cell

Edit distance

- How similar are two strings?

o c u r r a n c e –
o c c u r r e n c e
6 mismatches, 1 gap

o c – u r r a n c e
o c c u r r e n c e
1 mismatch, 1 gap

o c – u r r – a n c e
o c c u r r e – n c e
0 mismatches, 3 gaps

e.g. ocurrance and occurrence

Edit distance

- Edit distance between two strings:
  - gap penalty δ
  - mismatch penalty α_{pq}
  - distance = sum of gap + mismatch penalties

C T – G A C C T A C G
C T G G A C G A A C G
cost = $\delta + \alpha_{CG} + \alpha_{TA}$

  - many variations, many applications

String alignment

- Given two strings:
  - $x = x_1\ x_2 \dots x_m$
  - $y = y_1\ y_2 \dots y_n$

x_1 x_2 x_3 x_4 x_5 x_6
C T A C C – G
– T A C A T G
y_1 y_2 y_3 y_4 y_5 y_6

$M = \{(x_2, y_1), (x_3, y_2), (x_4, y_3), (x_5, y_4), (x_6, y_6)\}$

- alignment = sequence of pairs (x_i, y_j)
  - each symbol in at most one pair
  - no crossings: (x_i, y_j), $(x_{i'}, y_{j'})$ with $i < i'$, $j > j'$
  - cost(M) = $\sum_{(x_i,y_j)\in M} \alpha_{x_i,y_j} + \sum_{i:x_i \text{ unmatched}} \delta + \sum_{j:y_j \text{ unmatched}} \delta$

String alignment

- Given two strings:
  - $x = x_1\ x_2 \dots x_m$
  - $y = y_1\ y_2 \dots y_n$
- alignment = sequence of pairs (x_i, y_j)
  - cost(M)= $\sum_{(x_i,y_j)\in M} \alpha_{x_i,y_j} + \sum_{i:x_i \text{ unmatched}} \delta + \sum_{j:y_j \text{ unmatched}} \delta$
- Goal: find minimum cost alignment

String alignment

- subproblem: OPT(i, j) = minimum cost of aligning prefixes $x_1\ x_2 \dots x_i$ and $y_1\ y_2 \dots y_j$
  - case 1: x_i matched with y_j
    - cost = α_{x_i,y_j} + OPT(i-1, j-1)
  - case 2: x_i unmatched
    - cost = δ + OPT(i-1, j)
  - case 3: y_j unmatched
    - cost = δ + OPT(i, j-1)

String alignment

- subproblem: OPT(i, j) = minimum cost of aligning prefixes $x_1\ x_2 \ldots x_i$ and $y_1\ y_2 \ldots y_j$
- conclude:

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

String alignment

```
STRING-ALIGNMENT (m, n, x1, …, xm, y1, …, yn, δ, α)
FOR i = 0 TO m
    M[i, 0] ← iδ
FOR j = 0 TO n
    M[0, j] ← jδ
FOR i = 1 TO m
    FOR j = 1 TO n
        M[i, j] ← min { α[xi, yj] + M[i–1, j–1],
                        δ + M[i–1, j],
                        δ + M[i, j–1]}.
RETURN M[m, n].
```

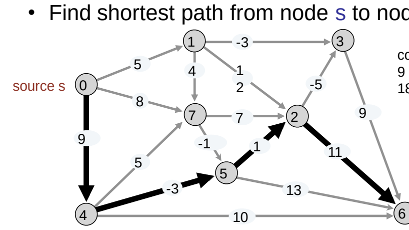

String alignment

```
STRING-ALIGNMENT (m, n, x1, …, xm, y1, …, yn, δ, α)
FOR i = 0 TO m
    M[i, 0] ← iδ
FOR j = 0 TO n
    M[0, j] ← jδ
FOR i = 1 TO m
    FOR j = 1 TO n
        M[i, j] ← min { α[xi, yj] + M[i–1, j–1],
                        δ + M[i–1, j],
                        δ + M[i, j–1]}.
RETURN M[m, n].
```

- running time?
  - O(nm)
- space?
  - O(nm)
- can improve to O(n + m) (how?)
- can recover alignment (how?)

Shortest paths (again)

- Given a directed graph G = (V, E) with (possibly negative) edge weights
- Find shortest path from node s to node t

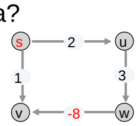


cost of path = 9 - 3 + 1 + 11 = 18

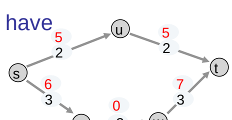
destination t

Shortest paths

- Didn't we do that with Dijkstra?
  - can fail if negative weights

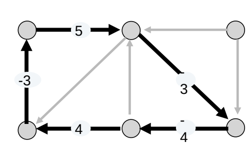


- Idea: add a constant to every edge?
  - comparable paths may have different # of edges



Shortest paths

- negative cycle = directed cycle such that the sum of its edge weights is negative

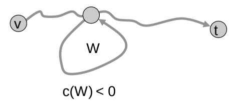


a negative cycle W : $c(W) = \sum_{e \in W} c_e < 0$

Shortest paths

Lemma: If some path from v to t contains a negative cycle, then there does not exist a shortest path from v to t

Proof: go around the cycle repeatedly to make path length arbitrarily small.



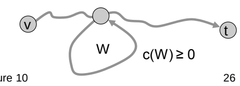
Shortest paths

Lemma If G has no negative cycles, then there exists a shortest path from v to t that is simple (has ≤ n – 1 edges)

Proof:

– consider a cheapest v↝t path P

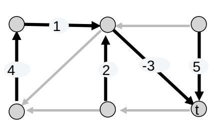
– if P contains a cycle W, can remove portion of P corresponding to W without increasing the cost



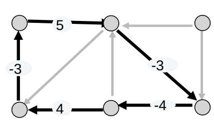
Shortest paths

Shortest path problem. Given a digraph with edge weights c_{vw} and no negative cycles, find cheapest v↝t path for each node v.

Negative cycle problem. Given a digraph with edge weights c_{vw}, find a negative cycle (if one exists).



shortest-paths tree



negative cycle

Shortest paths

- subproblem: OPT(i, v) = cost of shortest v↝t path that uses ≤ i edges
  - case 1: shortest v↝t path uses ≤ i – 1 edges
    - OPT(i, v) = OPT(i – 1, v)
  - case 2: shortest v↝t path uses i edges
    - edge (v, w) + shortest w↝t path using ≤ i -1 edges

$$OPT(i,v) = \begin{cases} \infty & \text{if } i=0 \\ \min\left\{ OPT(i-1,\, v),\ \min_{(v,w)\in E} \{ OPT(i-1,\, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

Shortest paths

- subproblem: OPT(i, v) = cost of shortest v↝t path that uses ≤ i edges

$$OPT(i,v) = \begin{cases} \infty & \text{if } i=0 \\ \min\left\{ OPT(i-1,\, v),\ \min_{(v,w)\in E} \{ OPT(i-1,\, w) + c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

- OPT(n-1, v) = cost of shortest v↝t path overall, if no negative cycles. Why?
  - can assume path is simple

Shortest paths

```
SHORTEST-PATHS (V, E, c, t)
FOREACH node v ∈ V
    M[0, v] ← ∞.
M[0, t] ← 0.
FOR i = 1 TO n – 1
    FOREACH node v ∈ V
        M[i, v] ← M[i – 1, v].
        FOREACH edge (v, w) ∈ E
            M[i, v] ← min { M[i, v], M[i – 1, w]
            + c_vw }.
```

Shortest paths

```
SHORTEST-PATHS (V, E, c, t)

FOREACH node v ∈ V
    M[0, v] ← ∞.
M[0, t] ← 0.
FOR i = 1 TO n – 1
    FOREACH node v ∈ V
        M[i, v] ← M[i – 1, v].
        FOREACH edge (v, w) ∈ E
            M[i, v] ← min { M[i, v],  M[i – 1, w] + c_vw }.
```

- running time? O(nm)
- space? $O(n^2)$
- can improve to O(n) (how?)
- can recover path (how?)

Shortest paths

- Space optimization: two n-element arrays
  - d(v) = cost of shortest v⤳t path so far
  - successor(v) = next node on current v⤳t path
- Performance optimization:
  - if d(w) was not updated in iteration i – 1, then no reason to consider edges entering w in iteration i

Bellman-Ford

```
BELLMAN-FORD (V, E, c, t)

FOREACH node v ∈ V
    d(v) ← ∞.
    successor(v) ← null.
d(t) ← 0.
FOR i = 1 TO n – 1
    FOREACH node w ∈ V
        IF (d(w) was updated in previous iteration)
            FOREACH edge (v, w) ∈ E
                IF ( d(v) > d(w) + c_vw)
                    d(v) ← d(w) + c_vw.
                    successor(v) ← w.
    IF no d(w) value changed in iteration i, STOP.
```

1 pass

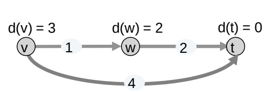
early stopping rule

Bellman-Ford

- notice that algorithm is well-suited to distributed, "local" implementation
  - n iterations/passes
  - each time, node v updates M(v) based on M(w) values of its neighbors
- important property exploited in routing protocols
- Dijkstra is "global" (e.g., must maintain set S)

Bellman-Ford

- Is this correct?
- Attempt: after the i^{th} pass, $d(v)$ = cost of shortest v⤳t path using at most i edges
  - counterexample:



if nodes w considered before node v, then d(v) = 3 after 1 pass

Bellman-Ford

<u>Lemma</u>: Throughout algorithm, $d(v)$ is the cost of some v⤳t path; after the i^{th} pass, $d(v)$ is no larger than the cost of the shortest v⤳t path using $\le i$ edges.

Proof (induction on i)

- Assume true after i^{th} pass.
- Let P be any v⤳t path with $i + 1$ edges.
- Let (v, w) be first edge on path and let P' be subpath from w to t.
- By inductive hypothesis, $d(w) \le c(P')$ since P' is a w⤳t path with i edges.
- After considering v in pass i+1: $d(v) \le c_{vw} + d(w) \le c_{vw} + c(P') = c(P)$

<u>Theorem</u>: Given digraph with no negative cycles, algorithm computes cost of shortest v⤳t paths in $O(mn)$ time and $O(n)$ space.

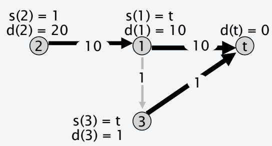
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



37

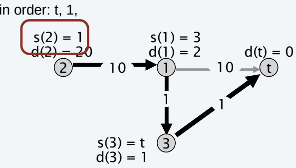
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.

consider nodes in order: t, 1, 2, 3



38

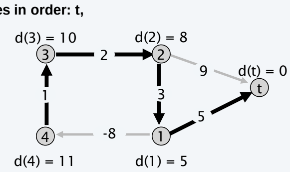
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



39

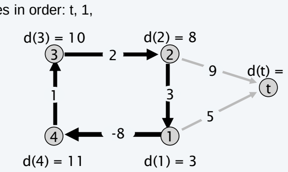
Bellman-Ford: analysis

Claim. ~~Throughout the Bellman-Ford algorithm, following *successor*(*v*) pointers gives a directed path from *v* to *t* of cost *d*(*v*).~~

Counterexample. Claim is false!

- Cost of successor $v \rightsquigarrow t$ path may have strictly lower cost than $d(v)$.
- Successor graph may have cycles.

consider nodes in order: t, 1, 2, 3, 4



40

Bellman-Ford

Lemma: If successor graph contains directed cycle W, then W is a negative cycle.

Proof:

- if $successor(v) = w$, we must have $d(v) \geq d(w) + c_{vw}$. (LHS and RHS are equal when $successor(v)$ is set; $d(w)$ can only decrease; $d(v)$ decreases only when $successor(v)$ is reset)
- Let $v_1 \to v_2 \to \ldots \to v_k$ be the nodes along the cycle W.
- Assume that (v_k, v_1) is the last edge added to the successor graph.
- Just prior to that:

$$\begin{aligned}
d(v_1) &\geq d(v_2) + c(v_1, v_2)\\
d(v_2) &\geq d(v_3) + c(v_2, v_3)\\
&\vdots\\
d(v_{k-1}) &\geq d(v_k) + c(v_{k-1}, v_k)\\
d(v_k) &> d(v_1) + c(v_k, v_1)
\end{aligned}$$

holds with strict inequality since we are updating $d(v_k)$

- add inequalities: $c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_1) < 0$

Bellman-Ford

Theorem: Given a digraph with no negative cycles, algorithm finds the shortest $s \rightsquigarrow t$ paths in $O(mn)$ time and $O(n)$ space.

Proof:

- The successor graph cannot have a cycle (previous lemma).
- Thus, following the successor pointers from s yields a directed path to t.
- Let $s = v_1 \to v_2 \to \ldots \to v_k = t$ be the nodes along this path P.
- Upon termination, if $successor(v) = w$, we must have $d(v) = d(w) + c_{vw}$. (LHS and RHS are equal when $successor(v)$ is set; $d(\cdot)$ did not change)
- Thus:

$$\begin{aligned}
d(v_1) &= d(v_2) + c(v_1, v_2)\\
d(v_2) &= d(v_3) + c(v_2, v_3)\\
&\vdots\\
d(v_{k-1}) &= d(v_k) + c(v_{k-1}, v_k)
\end{aligned}$$

since algorithm terminated

Adding equations yields $\underbrace{d(s)}_{\text{min cost of any } s \rightsquigarrow t \text{ path}} = \underbrace{d(t)}_{0} + \underbrace{c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k)}_{\text{cost of path } P}$