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)\(\cdot\) (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.
  
  \(x = \text{gctatgatctagcttatata}\)
  \(y = \text{catgacaagtgtcgactgtatctaaa}\)
  \(z = \text{tattctctta}\)
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 \( y_1 \ y_2 \ldots \ y_{n-1} \)

May 1, 2014 CS38 Lecture 10

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 \( y_1 \ y_2 \ldots \ y_{n-1} \)

May 1, 2014 CS38 Lecture 10

Longest common subsequence

• what order to fill in the table?

LCS-length(x, y: strings)
1. \( \text{OPT}(i, 0) = 0 \) for all \( i \)
2. \( \text{OPT}(0, j) = 0 \) for all \( j \)
3. for \( i = 1 \) to \( m \)
   4. for \( j = 1 \) to \( n \)
      5. if \( x_i = y_j \) then \( \text{OPT}(i, j) = \text{OPT}(i-1, j) + 1 \)
      6. else if \( \text{OPT}(i-1, j) \geq \text{OPT}(i, j-1) \) then \( \text{OPT}(i, j) = \text{OPT}(i-1, j) \)
      7. else if \( \text{OPT}(i, j-1) \geq \text{OPT}(i-1, j) \) then \( \text{OPT}(i, j) = \text{OPT}(i, j-1) \)
      8. \( \text{return} \left( \text{OPT}(m, n) \right) \)

May 1, 2014 CS38 Lecture 10

Longest common subsequence

• running time?
  - \( O(mn) \)

May 1, 2014 CS38 Lecture 10

Longest common subsequence

• space \( O(nm) \)
  - can be improved to \( O(\min(n,m)) \)

May 1, 2014 CS38 Lecture 10
Longest common subsequence

LCS-length(x, y: strings)
1. \( \text{OPT}(i, 0) = 0 \) for all \( i \)
2. \( \text{OPT}(0, j) = 0 \) for all \( j \)
3. for \( i = 1 \) to \( m \)
4. for \( j = 1 \) to \( n \)
5. if \( x_i = y_j \) then \( \text{OPT}(i, j) = \text{OPT}(i-1, j-1) + 1 \)
6. else \( \text{OPT}(i, j) = \max(\text{OPT}(i-1, j), \text{OPT}(i, j-1)) \)
7. end for
8. return(\( \text{OPT}(m, n) \))

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

Edit distance

• How similar are two strings?
  – gap penalty \( \delta \)
  – mismatch penalty \( \alpha_{pq} \)
  – distance = sum of gap + mismatch penalties
  – many variations, many applications

String alignment

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

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

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

Shortest paths (again)

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

$$\begin{align*}
\text{cost of path } & = 9 \cdot 3 = 1 + 11 = 18 \\
\end{align*}$$

String alignment

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

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

String alignment

$$\text{STRING-ALIGNMENT}\left(\text{m}, \text{n}, x_1, \ldots, x_m, y_1, \ldots, y_n, \delta, \alpha, \kappa\right)$$

- For $i = 0$ to $m$
  - $M[i, 0] \leftarrow i\delta$
- For $j = 0$ to $n$
  - $M[0, j] \leftarrow j\delta$
- For $i = 1$ to $m$
  - For $j = 1$ to $n$
    - $M[i, j] \leftarrow \min \{ \alpha(x_i, y_j) + M[i-1, j-1], \delta + M[i-1, j], \delta + M[i, j-1] \}$
- RETURN $M[m, n]$. 

Shortest paths

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

- a negative cycle $W$: $\epsilon[W] = \sum_{e_i} e_i < 0$

String alignment

- 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

**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.

\[
\begin{align*}
\text{c(W)} &< 0
\end{align*}
\]

May 1, 2014 CS38 Lecture 10

---

Shortest paths

**Negative cycle problem**. Given a digraph with edge weights \( c_{vw} \) and no negative cycles, find cheapest \( v \to t \) path for each node \( v \).

\[
\begin{align*}
\text{OPT}(i, v) &= \begin{cases} 
\min \{ \text{OPT}(i-1, v), \ \min_{0 < w \in V} (\text{OPT}(i-1, w) + c_{vw}) \} & \text{if } i > 0 \\
\infty & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{OPT}(n, v) &= \begin{cases} 
\min \{ \text{OPT}(1, v), \ \min_{0 < w \in V} (\text{OPT}(1, w) + c_{vw}) \} & \text{if } i = 0 \\
\infty & \text{otherwise}
\end{cases}
\end{align*}
\]

May 1, 2014 CS38 Lecture 10

---

Shortest paths

**subproblem**: \( \text{OPT}(i, v) = \text{cost of shortest } v \to t \text{ path that uses } \leq i \text{ edges} \)

\[
\begin{align*}
\text{OPT}(i, v) &= \begin{cases} 
\min \{ \text{OPT}(i-1, v), \ \min_{0 < w \in V} (\text{OPT}(i-1, w) + c_{vw}) \} & \text{if } i > 0 \\
\infty & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{OPT}(n, v) &= \begin{cases} 
\min \{ \text{OPT}(1, v), \ \min_{0 < w \in V} (\text{OPT}(1, w) + c_{vw}) \} & \text{if } i = 0 \\
\infty & \text{otherwise}
\end{cases}
\end{align*}
\]

May 1, 2014 CS38 Lecture 10

---

Shortest paths

**Lemma**: If \( G \) has no negative cycles, then there exists a shortest path from \( v \) to \( t \) that is simple (has \( \leq n - 1 \) edges).

Proof:

- consider a cheapest \( v \to t \) path \( P \)
- if \( P \) contains a cycle \( W \), can remove portion of \( P \) corresponding to \( W \) without increasing the cost

\[
\begin{align*}
\text{OPT}(v, t) &= \min \{ \text{OPT}(i, v), \ \min_{0 < w \in V} (\text{OPT}(i-1, w) + c_{vw}) \} & \text{if } i = 0 \\
\infty & \text{otherwise}
\end{align*}
\]

May 1, 2014 CS38 Lecture 10

---

**Shortest-Path (V, E, c, i)**

**FOR EACH** node \( v \in V \)

\[
\begin{align*}
M[0, v] &\rightarrow \infty \\
M[0, t] &\rightarrow 0
\end{align*}
\]

**FOR** \( i = 1 \) **TO** \( n - 1 \)

**FOR EACH** node \( v \in V \)

\[
\begin{align*}
M[i, v] &\rightarrow M[i-1, v] \\
M[i, v] &\rightarrow \min \{ M[i, v], M[i-1, w] + c_{wv} \}
\end{align*}
\]

May 1, 2014 CS38 Lecture 10
Shortest paths

Shortest-Paths (V, E, c, t)

FOR EACH node v ∈ V
  M[v, v] ← 0,
  M[v, v] ← ∞.

FOR i = 1 TO n - 1
  FOR EACH node v ∈ V
    M[v, v] ← M[i-1, v].
    FOR EACH edge (v, w) ∈ E
      M[v, w] ← min { M[v, v], M[i-1, w] + c vw }.

performance optimization:

• running time? O(nm)
• space? O(n²)
• can improve to O(n) (how?)
• can recover path (how?)

Bellman-Ford

Bellman-Ford (V, E, c, t)

FOR EACH node v ∈ V
  d(v) ← ∞,
  predecessor(v) ← null.
  d(t) ← 0.

FOR i = 1 TO n - 1
  FOR EACH node v ∈ V
    If (d(v) was updated in previous iteration)
      FOR EACH edge (v, w) ∈ E
        If (d(v) > d(w) + c vw )
          d(v) ← d(w) + c vw.
        predecessor(v) ← w.
      
      If no d(v) value changed in iteration i, STOP.

May 1, 2014 CS38 Lecture 10 31

May 1, 2014 CS38 Lecture 10 32

May 1, 2014 CS38 Lecture 10 34

May 1, 2014 CS38 Lecture 10 35

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:

  d(v) = 3
d(w) = 2
  d(t) = 0

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

May 1, 2014 CS38 Lecture 10 36

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)

Lemma: 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 ≤ i edges.

Proof (induction on i)
  — Assume true after i-1 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) ≤ c(P’) since P’ is a w→t-path with ≤ i edges.
  — After considering v in pass i + 1:
    d(v) ≤ c(v, w) + c(P’)
    = c(P)

Theorem: 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 u to z of cost c(v).

Counterexample. Claim is false!
* Cost of successor v→ path may have strictly lower cost than d(v).
Consider nodes in order: t, 1, 2, 3
\[d(1) = 10 \quad d(2) = 8 \quad d(3) = 10 \quad d(4) = 11\]

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; \( d(v) \) can only decrease; \( d(w) \) decreases only when successor(v) is reset)
- Let \( v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \) be the nodes along the cycle \( W \).
- Assume that \( (v_{k-1}, v_k) \) is the last edge added to the successor graph.
- Just prior to that:
  \[d(v_1) \geq d(v_2) + c_{v_1, v_2}\]
  \[d(v_2) \geq d(v_3) + c_{v_2, v_3}\]
  \[\vdots\]
  \[d(v_{k-1}) \geq d(v_k) + c_{v_{k-1}, v_k}\]
- \( d(v_k) > d(v_{k-1}) + c_{v_{k-1}, v_k} \)
- add inequalities: \( c(v_1, v_2) + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) + c(v_k, v_{k-1}) < 0 \)

May 1, 2014
CS358 Lecture 10

Bellman-Ford: analysis

Claim. Throughout the Bellman-Ford algorithm, following-successor(v) pointers gives a directed path from u to z of cost c(v).

Counterexample. Claim is false!
* Cost of successor v→ path may have strictly lower cost than d(v).
Consider nodes in order: t, 1, 2, 3
\[s(2) = 1 \quad d(2) = 20 \quad s(1) = t \quad d(1) = 10 \quad d(3) = 1\]

Bellman-Ford

Theorem: Given a digraph with no negative cycles, algorithm finds the shortest u→f 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 \( u \) yields a directed path to \( t \).
- Let \( u \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow f \) 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; \( d(v) \) did not change)
- Thus:
  \[d(v_1) = d(v_2) + c_{v_1, v_2}\]
  \[d(v_2) = d(v_3) + c_{v_2, v_3}\]
  \[\vdots\]
  \[d(v_{k-1}) = d(v_k) + c_{v_{k-1}, v_k}\]
Adding equations yields: \( d(v) = d(v_1) + c_{v_1, v_2} + c(v_2, v_3) + \ldots + c(v_{k-1}, v_k) \)

Bellman-Ford: analysis

Claim. Throughout the Bellman-Ford algorithm, following-successor(v) pointers gives a directed path from u to z of cost c(v).

Counterexample. Claim is false!
* Cost of successor v→ path may have strictly lower cost than d(v).
Consider nodes in order: t, 1, 2, 3
\[s(2) = 1 \quad d(2) = 10 \quad s(3) = t \quad d(3) = 1\]