CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- SA4, SA5 due already
- Full PS4 out, due Nov. 30
  - No exercises beyond the Lookahead
- Small modification to Late Problem Set Policy from Syllabus:
  - If you have 3 PSs submitted more than 1 week late, any additional PSs submitted more than 1 week late will be a 35% deduction
    - Please turn in PS4 and PS5 on time!
- Project 3 out, due 11:59pm Monday, Nov. 21
- PS3 grading update
- Project 4
  - Will be out next Monday, Nov. 28
  - Will be due no sooner than 2 weeks after that
  - Intended team size: 4 (but talk to me if you'd prefer to work with a smaller team size)

For the Most Part…

- The efficient Fibonacci methods used a characteristic technique of dynamic programming:
  - Results stored in a table (or similar), used to improve efficiency
- Dynamic programming solutions can be either top-down or bottom-up…
  - But most of the time, in practice, when people talk about a dynamic programming solution, they mean a bottom-up solution
- In general, when looking for a dynamic programming solution:
  - Try recursive, top-down approach with overlapping sub-problems
  - (Consider a memoized version)
  - Then, try bottom-up, iterative approach based on sub-problems
  - (Then, try to improve on space complexity of bottom-up method)

For the Most … Part 2

- Dynamic programming is often applied to *optimization problems*, to find a solution with an optimal (minimal or maximal) value
  - Often, for optimization problems, it is (or seems) necessary to consider all subsets of a set
  - … so, if we're looking at a set of size n, what's the time complexity of such an algorithm?
- Characteristic structure for dynamic programming algorithms
  - Overlapping subproblems (as previously seen)
  - *Optimal substructure*: an optimal solution is built from the optimal solutions of subproblems

This makes total sense if you think about it for a while! If there isn't redundant work in the algo, or if optimal solutions aren't based on optimal solutions to subproblems, then why would we store solutions to subproblems?

- Steps in developing a dynamic programming algorithm:
  1. Characterize the structure of an optimal solution (in words)
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution from the bottom up
  4. Construct an optimal solution from computed information

These are on CLRS, pg. 359

We'll focus on steps 2 and 3, leading to step 4

Bottom-up Computation of Optimal LCS Value

- Need m-by-n matrix C to store lengths:
  - To compute C[i,j], need values of C[i-1,j-1] (when $x_i = y_j$) and C[i-1, j] and C[i, j-1] (when $x_i \neq y_j$)

Actually (m+1)-by-(n+1), to include the 0 case, too

Recall our recursive definition: Base case: C[0,j] = 0 and C[i,0] = 0 for all i, j
Recursive step, to compute C[i,j] for i,j > 0: If $x_i = y_j$, C[i,j] = C[i-1,j-1] + 1
If $x_i \neq y_j$, C[i,j] = max(C[i,j-1], C[i-1,j])

```
LCS(X, Y) // input: sequences X, Y
m ← length(X)
n ← length(Y)
for i ← 0 to m do  C[i, 0] ← 0  // 0 in first col of each row
for j ← 0 to n  do  C[0, j] ← 0 // 0 in first row of each col
for i ← 1 to m do
    for j ← 1 to n do    // process row by row
        if x_i = y_j then C[i, j] ← C[i-1, j-1] + 1
        else C[i, j] ← max (C[i, j-1], C[i-1, j])
return C[m, n]
```

- What is the time complexity of this algorithm?

- What is the optimal length—the length of an LCS of full sequences X and Y?

Bottom-up Computation of An Optimal LCS

- To find an LCS, also store which symbols (indices of symbols) are actually part of the LCS as it's being built
  - i.e., which table elements have optimal sub-problem values
  - if $x_i = y_j$, answer came from the upper left (diagonal) of current element
    (i.e., one less elt. of *both* X and Y)
  - if $x_i \neq y_j$ the answer came from above or to the left, whichever is larger (if equal, we can choose "above", by convention)
    (i.e., …*either* X or Y)

```
LCS(X, Y)
    m ← length(X)
    n ← length(Y)
    for i ← 0 to m do  C[i, 0] ← 0
    for j ← 0 to n  do  C[0, j] ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if x_i = y_j then C[i, j] = C[i-1, j-1] + 1
                B[i, j] ← Up&Left
            else
                if C[i - 1, j]  >= C[i, j - 1] then
                    C[i, j] ← C[i - 1, j]
                    B[i, j] ← Up //one less elt. of X
                else C[i , j] ← C[i, j - 1]
                    B[i, j] ← Left //one less elt. of Y
```

- LCS example:

LCS of
X = <A,B,C,B,D,A,B>
and
Y = <B,D,C,A,B,A>

The algorithm finds an LCS: *BCBA*

Are there others?

	j	0	1	2	3	4	5	6
i		y_j	*B*	*D*	*C*	*A*	*B*	*A*
0	x_i	0	0	0	0	0	0	0
1	*A*	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	*B*	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	*C*	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	*B*	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	*D*	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	*A*	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	*B*	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

And finally…
Finding a Solution from the Values

- That bottom-up method gives us the information from which we can get an optimal value and the associated indices
- To actually find / print the longest common subsequence, start at the bottom-left of the table and follow the arrows:

Print- LCS (B,X,i,j)

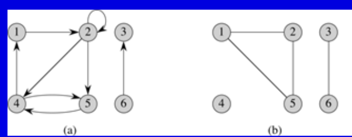
1. if i = 0 or j = 0 then return
2. if B[i,j] = Up&Left
3. then Print-LCS(B,X,i-1,j-1)
4. print x_i
5. else if B[i,j] = Up
6. then Print-LCS(B,X,i-1,j)
7. else Print-LCS(B,X,i,j-1)

Initial call has
i = m (i.e., length(X)),
j = n (length(Y))

B is the "arrow table" from the previous slide

Graphs

(See Sec. B.4)



(a) Directed graph
(b) Undirected graph

- Graphs commonly represent connections among related elements
- An *undirected* graph G = (V,E) is
  - A set V of vertices (nodes), and
  - A set E of edges (linked pairs of nodes), which are bidirectional
- A *directed* graph (or *digraph*) G = (V,E) is
  - A set V of vertices (nodes), and
  - A set E of unidirectional edges (typically represented as arrows). Note that self-loops—edges from a node to itself—are possible
- Convention: For algorithm analysis, we may use V for |V| and E for |E|

• So, in undirected graphs, edges are *unordered* pairs of vertices, whereas in digraphs, edges are *ordered* pairs.

• By convention, in an undirected graph G = (V,E), we consider (u,v) and (v,u) to be the same edge, so at most one of those pairs will be in E.

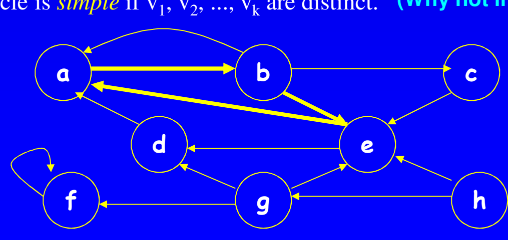
More Graph Vocabulary

So we can talk about more graphs

- Graphs with the same number of vertices can have different numbers of edges. What's the most edges a graph can have, in terms of |V|?
  - A *sparse* graph is one in which |E| is much less than $|V|^2$
  - A *dense* graph is one in which |E| is close to $|V|^2$

(See CLRS, pg. 589)

- In a digraph, a path $(v_0, v_1, ..., v_k)$ forms a *cycle* if $v_0 = v_k$ and the path contains at least one edge
  - A cycle is *simple* if $v_1, v_2, ..., v_k$ are distinct. (Why not include v_0 in this?)

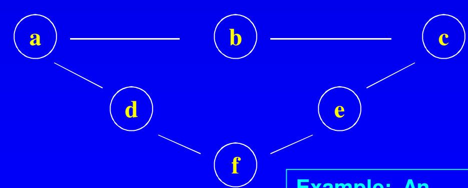


- A directed graph with no cycles is a *directed acyclic graph*, or *DAG*

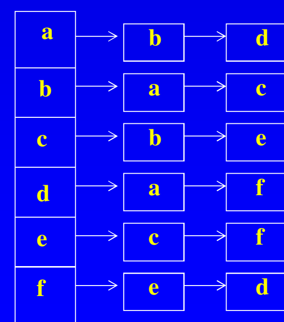
(Read CLRS Sec. 22.1-22.3)

Adjacency List Representations

- A graph can be represented as an *adjacency list*
  - For each vertex v, there's a list of all nodes adjacent to v
  - Represented as an array of |V| = n lists



Example: An undirected graph

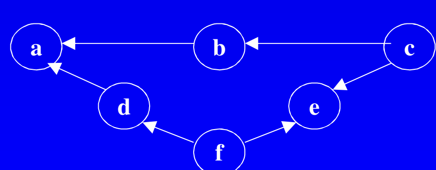


- Complexity:
  - Storage space: O(V + E)
  - (For what kinds of graphs is this an efficient storage representation?)
  - What's the time complexity to find if an edge is in a graph?

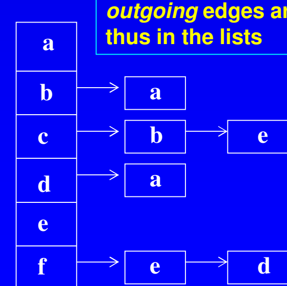
Adjacency List Representations

- A graph can be represented as an *adjacency list*
  - For each vertex v, there's a list of all nodes adjacent to v
  - Represented as an array of |V| = n lists

For digraphs, only *outgoing* edges are in E, thus in the lists



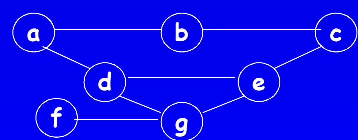
Example: a digraph



“There is no spoon next to anything” – from *The Adjacency Matrix*

Graphs: Adjacency Matrix Representations

- A graph can be represented as an *adjacency matrix*
  - A V-by-V matrix : For each pair (i,j) of vertices, entry (i,j) in the matrix is 1 if (i,j) \in E, and 0 otherwise



Example: An undirected graph

	a	b	c	d	e	f	g
a	0	1	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	0	1	0	0
d	1	0	0	0	1	0	1
e	0	0	1	1	0	0	1
f	0	0	0	0	0	0	1
g	0	0	0	1	1	1	0

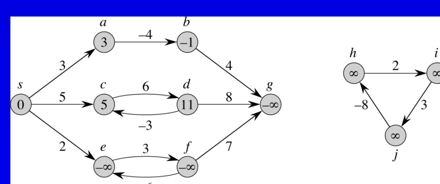
- Complexity:
  - Storage space: $O(V^2)$
  - What’s the time complexity to find if an edge is in a graph?

For directed graphs, only outgoing edges are in E (thus in the matrix)

“Weighted Graphs and Shortest Paths” could be the album title for a collaboration between xkcd and Death Cab For Cutie.

Well, it *could* be.

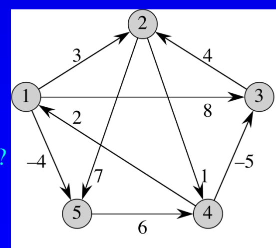
Weighted Graphs and Shortest Paths



- Graphs can be *weighted*:
  - Given G=(V,E), there is a weight function *w* that maps each edge in E to a real-valued *weight*
  - For weighted graphs, an adjacency matrix can store the weight of an edge in E (rather than just 0 or 1)
- Given such a *w*, we say the weight of a path *w*(p) for p = $<v_0, v_1, \ldots, v_k>$ is the sum of the weights of the edges on that path (i.e., (v_0, v_1), (v_1, v_2), … (v_{k-1}, v_k))
- We then say the *shortest path weight* δ(u,v) from vertex u to vertex v is the least weight of any path in G from u to v
  - A shortest path from u to v is any path from u to v in G with that weight

Shortest Path Problems

- Kinds of graph problems based on finding shortest paths (by convention, presume weighted, directed graphs):
  - Single-source shortest paths
    - Various algorithms for cases of it (e.g., *Dijkstra*)
  - Single-destination shortest paths
    - If we have a single-source shortest paths algorithm, how could we solve this?
  - Single-pair shortest path
    - How does this relate to the single-source variant?
  - **All-pairs shortest paths**
    - **We'll talk more about this soon**



- (Note: To represent a (shortest) path in solving such a problem, each vertex is presumed to have a predecessor field, which stores its predecessor on the path being considered.)

Properties of Shortest Paths

- Optimal substructure of shortest paths
  - Is each sub-path of a shortest path itself a shortest path?
  - What's the argument for / counter-argument to that?
- Can a shortest path in a weighted graph have a cycle?
  - (Be sure to consider graphs with negative edges, which could have negative weight cycles, as well as graphs with positive weight cycles!)

All-Pairs Shortest Paths

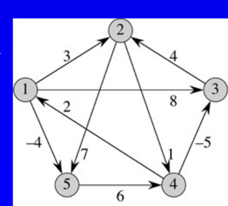
- The All-Pairs Shortest Paths Problem:
  - Given weighted graph G=(V, E) (with no negative weight cycles), find the shortest path from u to v for every u, v \in V
- Solutions can be based on dynamic programming and an adjacency matrix representation of G
  - Recall: Adjacency matrix W contains weight of each edge in E
  - By convention, diagonal of W is all 0s
- How might we break this down into sub-problems for a recursive solution?

All-Pairs Shortest Paths: A Vertex-Based Recursive Solution

- Solve all-pairs shortest path problem in terms of the *intermediate* vertices that can appear on any shortest path
  - *Intermediate vertex* of a *simple* path $p = <v_1, v_2, \ldots, v_z>$ is any vertex on p other than v_1 or v_z

A *simple* path is a path with all distinct vertices

- For graph G, call vertices V = {1, …, n}, and consider subsets V_k = {1, … , k} of V
- Then, for any two vertices i, j in V, consider all paths from i to j with intermediate vertices drawn only from V_k
  - In particular, consider a shortest path p from i to j with intermediate vertices in V_k
  - What's the relationship between p and the set of shortest paths from i to j with intermediate vertices in V_{k-1}?



Also, is p a simple path? How do we know, one way or another?

All-Pairs Shortest Paths: A Vertex-Based Recursive Solution

- … We're still considering shortest path p from i to j with intermediate vertices in V_k
  - What's the relationship between p and the set of shortest paths from i to j with intermediate vertices in V_{k-1}?
- … Depends on whether or not vertex k is an intermediate vertex on path p
  - If not, then p is also a shortest path (i to j) with intermediate vertices in V_{k-1}
  - If so, then p can be broken down into sub-paths that are shortest paths with intermediate vertices in V_{k-1}
  - … one sub-path is from (i to k), the other is from (k to j) $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$

    How do we know we can decompose p that way, i.e., that both sub-paths are shortest paths, using only vertices numbered up to k-1?
  - Given this, how could we recursively define the shortest path lengths between all pairs of vertices?

All-Pairs Shortest Paths: A Vertex-Based Recursive Solution

- … We're still considering shortest path p from i to j with intermediate vertices in V_k
  - What's the relationship between p and the set of shortest paths from i to j with intermediate vertices in V_{k-1}?
- … Depends on whether or not vertex k is an intermediate vertex on path p
  - If not, then p is also a shortest path (i to j) with intermediate vertices in V_{k-1}
  - If so, then p can be broken down into sub-paths that are shortest paths with intermediate vertices in V_{k-1}
  - … one sub-path is from (i to k), the other is from (k to j) $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$
- Altogether, if W is the weights matrix, and $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k…

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

"By the way, which one's Warshall?"

Floyd-Warshall Algorithm: Bottom-up All-Pairs Shortest Paths

- *Floyd-Warshall algorithm* for all-pairs shortest paths: the bottom-up method based on this decomposition
- Computes matrices $D^{(k)} = (d_{ij}^{(k)})$, where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

Note: This computes shortest path values, not the paths. See CLRS pages 695-697 about computing the paths themselves.

- What does this algorithm return? (What makes that a useful return value?)
- What is the running time of this algorithm?

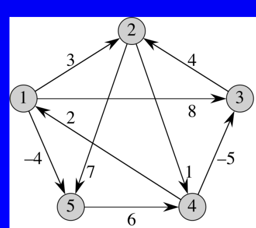
A Floyd-Warshall Example

See CLRS, Ch. 25.2

- Computes matrices D(k) = ($d_{ij}^{(k)}$), where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

- What D matrices does it compute for this example graph?

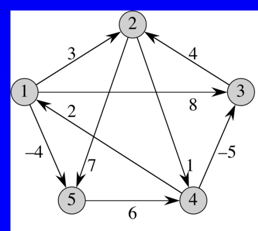


A Floyd-Warshall Example

- Computes matrices D(k) = ($d_{ij}^{(k)}$), where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

- What D matrices does it compute for this example graph?



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

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

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

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

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

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