CPSC 311 Lecture Notes

Graph Algorithms, Disjoint Sets ADTs, and Dynamic Programming
(Chapters 21, 22-25, 15)

Acknowledgement: Parts of these course notes are based on notes from courses given by Jennifer Welch at Texas A&M University.
Graph Algorithms - Outline of Topics

Topic Outline

• Elementary Graph Algorithms – Chapt 22 (review)
  – graph representation
  – depth-first-search, breadth-first-search, topological sort

• Union-Find Data Structure – Chapt 21
  – implementing dynamic sets

• Minimum Spanning Trees – Chapt 23
  – Kruskal’s and Prim’s algorithms (greedy algorithms)

• Single-Source Shortest Paths – Chapt 24
  – Dijkstra’s algorithm (greedy)

• Dynamic Programming – Chapt 15
  – All Pairs Shortest Paths (Chapt 25, Floyd-Warshall Alg as example)
Graphs

A graph $G = (V, E)$ consists of:

- a set $V$ of nodes (vertices), and
- a set $E$ of edges (pairs of nodes)
  - directed or undirected
- for analysis, we may use $V$ for $|V|$ and $E$ for $|E|$  
  - or more often, $n = |V|$ and $m = |E|$
Representing a Graph – Adjacency List

**Adjacency List:**
An array $A[1, n]$ of lists, one for each node $v \in V$.
$v$’s list contains (ptrs to) all nodes adjacent to $v$ in $G$

**example:**

![Graph Diagram]

<table>
<thead>
<tr>
<th>Node</th>
<th>Adjacencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3</td>
</tr>
<tr>
<td>2</td>
<td>1, 3, 4</td>
</tr>
<tr>
<td>3</td>
<td>2, 5</td>
</tr>
<tr>
<td>4</td>
<td>1, 7</td>
</tr>
<tr>
<td>5</td>
<td>3, 7, 4</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>5, 6, 4</td>
</tr>
</tbody>
</table>

**Complexity Issues**
- advantage – storage is $O(n + m)$ (good for **sparse** graphs)
- drawback – need to traverse list to check for an edge
- advantage – find a vertex’s incident edges by traversing its adjacency list
  (the time depends on vertex’s **degree**, i.e., the number of incident edges)
Representing a Graph – Adjacency Matrix

**Adjacency Matrix:**
An array $A[n, n]$ such that

$$A[i, j] = \begin{cases} 
1 & \text{if } (i, j) \in E \\
0 & \text{otherwise} 
\end{cases}$$

example:

![Graph Diagram]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Complexity Issues**

- drawback – storage is $O(n^2)$ (good for **dense** graphs)
- advantage – $O(1)$ time to check for an edge
- drawback – need to check entire row (column) to find all incident edges (time is always $O(n)$, does not depend on the vertex degree)
Breadth-First Search

BFS\((G, s)\)
   enqueue\((Q, s)\) and visit\((s)\)
   while (not-empty\(Q\))
      \(u := \text{dequeue}(Q)\)
      for each \(v\) adjacent to \(u\)
         if (\(v\) not visited yet)
            enqueue\((Q, v)\) and visit\((v)\)
      endfor
   endwhile

Complexity (Adjacency List Representation)

- each node enqueued and dequeued once – \(O(n)\) time
- each edge considered once (in each direction) – \(O(m)\) time
- total = \(O(n + m)\)

Note: If \(G = (V, E)\) is not connected, then BFS will not visit the entire graph (without an external driver that makes sure all vertices are eventually visited – need to call BFS iteratively with an unvisited vertex \(s\))
Exercise: Breadth-First Search and Adjacency Matrices

\[
\text{BFS}(G, s)
\begin{align*}
\text{enqueue}(Q, s) \text{ and visit}(s) \\
\text{while } (\text{not-empty}Q) \\
\quad u := \text{dequeue}(Q) \\
\quad \text{for each } v \text{ adjacent to } u \\
\quad \quad \text{if } (v \text{ not visited yet}) \\
\quad \quad \quad \text{enqueue}(Q, v) \text{ and visit}(v) \\
\quad \text{endfor} \\
\text{endwhile}
\end{align*}
\]

Exercise:

1. What is the running time of BFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?
Depth-First Search

\[
\text{DFS}(G, s) \\
\text{visit}(s) \\
\text{for each } v \text{ adjacent to } s \\
\text{if } (v \text{ not visited yet}) \\
\text{DFS}(G, v) \\
\text{endfor}
\]

Complexity (Adjacency List Representation)

- check all edges adjacent to each node encountered \(- O(m) \text{ time}\)
- need to do ’clean up’ to deal with directed or unconnected graphs
- \( \text{total} = O(n + m) \)

**Note:** DFS divides edges into two categories:

- **tree edges** (edges traversed in DFS)
- **back edges** (other edges)

**Claim:** if \((u, v)\) is a back edge, then \(v\) is an ancestor of \(u\)
Depth-First Search

\[
\text{DFS}(G, s) \\
\text{visit}(s) \\
\text{for each } v \text{ adjacent to } s \\
\quad \text{if } (v \text{ not visited yet}) \\
\quad \text{DFS}(G, v) \\
\text{endfor}
\]

Exercise:

1. What is the running time of DFS if its input graph is represented by an adjacency matrix and the algorithm is modified to handle this form of input?
Topological Sort (application of DFS)

input: directed acyclic graph (DAG)
output: ordering of nodes st if \((u, v)\) in \(E\), then \(u\) comes before \(v\) in ordering

\textbf{TOPOLOGICAL-SORT}(\(G\))

1. do modified DFS, recording nodes in reverse order in which DFS finishes
   examining adjacency lists

Complexity (Adjacency List Representation) – \(O(n + m)\)

diagram:

DFS Discovery Order: a, b, c, h, i, f, d, e, g, j

Reverse Completion Order: g, j, a, d, e, b, f, c, h, i
Data Structures for Disjoint Sets

Disjoint Set Abstract Data Type (ADT) (Chapt 22)

- set of underlying elements $U = \{1, 2, \ldots, n\}$
- collection of disjoint subsets $C = s_1, s_2, \ldots, s_m$ of $U$
  - each $s_i$ has an id (a distinguished element)
- operations on $C$
  - $\text{create}(x)$: $x \in U$, makes set $\{x\}$
  - $\text{union}(x, y)$: $x, y \in U$ and are id’s of their resp. sets $s_x$ and $s_y$, replaces sets $s_x$ and $s_y$ with a set that is $s_x \cup s_y$ and returns the id of the new set
  - $\text{find}(x)$: $x \in U$, returns the id of the the set containing $x$

Note 1: The $\text{create}$ operation is typically used during the initialization of a particular algorithm.

Note 2: We assume we have a pointer to each $x \in U$ (so we never have to look for a particular element). Thus the the problems we’re trying to solve are how to manage the sets ($\text{unions}$) and how to find the id of the set containing a particular element ($\text{finds}$)
Graph Algorithm Application – Connected Components

\[
\text{CONNECTED-COMPONENTS}(G)
\]

\[
\text{let } U = V = \{a, b, c, d, e, f, g\}
\]

1. for each \( v \in U \)
   create\( (v) \)

2. for each \((u, v) \in E \)
   if find\( (u) \neq \text{find} (v) \)
   union\( (\text{find}(u), \text{find}(v)) \)

Initially (after step 1), we have the \( V \) sets:

\[
\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}
\]

Finally (after step 2), we have the two sets:

\[
\{a, b, c, d\}, \{e, f, g\}
\]

which represent the two connected components of the graph.
Simple Forest Implementation

**Idea:** organize elements of each set as a tree with id = root element, and a pointer from every child to its parent (recall we have pointers to every element)

**find**(*x*):
- start at *x* (using pointer provided) and follow pointers up to the root, and return id of root
- worst-case running time is \(O(n)\), where \(n = |U|\)

**union**(*x*, *y*):
- *x* and *y* are ids (roots of trees)
- make *x* a child of *y* and return *y*
- running time is \(O(1)\)

**Claim:** Amortized (average) worst-case cost per operation is \(O(n)\)

eexample worst-case sequence of \(2n - 1\) operations:

\[ S = U(1, 2), U(2, 3), U(3, 4), \ldots, U(n - 1, n), F(1), F(2), F(3), \ldots, F(n) \]

- total cost for \(n\) unions: \(O(n)\)
- total cost for \(n\) finds: \(n + (n - 1) + \ldots + 1 = O(n^2)\)
- average cost per operation: \(O(n^2)/2n = O(n)\)
Weighted Union Implementation

Idea: add weight field to each node holding the number of nodes in the subtree rooted at this node (we’ll only care about weight field of roots)

\textbf{find}(x):

- as before with simple implementation, running time $O(\log n)$

\textbf{union}(x, y):

- $x$ and $y$ are ids (roots of trees)
- make node ($x$ or $y$) with smaller weight the child of the other

\textbf{Theorem:} Any $k$-node tree created by $k - 1$ weighted unions has height $\leq \log k$ (assume we start with singleton sets). That is, trees stay ‘short’.

\textbf{Proof:} By induction on $k$.

\textbf{Basis:} $k = 1$, height $= 0 = \log 1$

\textbf{Inductive Hypothesis:} Assume true for all $i < k$.

\textbf{Induction Step:} Show true for $k$. Suppose the last operation performed was union($x, y$) and that $m = \text{wt}(x) \leq \text{wt}(y) = k - m$, so that $m \leq k/2$.

\[ h = \max(h_x + 1, h_y) \leq \log k \]

- $h_x + 1 \leq \log m + 1 \leq \log \frac{k}{2} + 1 = \log k - 1 + 1 = \log k$
- $h_y \leq \log(k - m) \leq \log k$
Weighted Union Implementation: Exercise

\textbf{find}(x):

- start at $x$ (using pointer provided) and follow pointers up to the root, and return id of root
- worst-case running time is $O(n)$, where $n = |U|$

\textbf{union}(x, y):

- $x$ and $y$ are ids (roots of trees)
- each set has a weight = the number of elements in the set
- make node ($x$ or $y$) with smaller weight the child of the other

\textbf{Exercises}

1. Consider the following sequence $S$ of $n - 1$ operations, where $F(i)$ denotes $\text{find}(i)$ and $U(j, k)$ denotes $\text{union}(j, k)$, where $i, j$ and $k$ denote elements $i, j,$ and $k$. Assume we use normal finds and the weighted unions (as described above).

$$S = U(1, 2), U(2, 3), U(3, 4), \ldots, U(n - 1, n), F(1), F(2), F(3), \ldots, F(n)$$

(a) total cost for $n - 1$ unions: ____________
(b) total cost for $n$ finds: ____________
(c) average cost per operation: ____________
Path Compression Implementation

**Idea:** extend the idea of weighted union (i.e., unions still weighted), but on a `find(x)` operation, make every node on the path from `x` to the root (the id) a child of the root.

So... `find(x)` still has worst-case time of $O(\log n)$, **but** subsequent `finds` for nodes that used to be ancestors of `x` will now be very fast.

**Theorem:** Let $S$ be any sequence of $O(n)$ unions and finds. Then the worst-case time for performing $S$ with weighted unions and path compression is $O(n \log^* n)$

$\log^* n$ is ‘almost’ constant

$\log^* n$ is the number of times we have to take the log of a number to reach 1:

- $\log^* 2 = 1$ ($\log^* 2 = 1$)
- $\log^* 3 - \log^* 4 = 2$ ($\log^* 2^2 = 2$)
- $\log^* 16 = 3$ ($\log^* 2^4 = 3$)
- $\log^* 65536 = 4$ ($\log^* 2^{2^2} = 4$)
Exercise: Union/Find Data structure

for i := 1 to 16
  makeset(i)
for i := 1 to 15
  union( find(i), find(i+1) )
for i := 1 to 16
  find(i)

Exercises:

1. Draw the final data structure that results from applying the above sequence of operations when **neither** weighted union nor path compression are used.

2. Draw the final data structure that results from applying the above sequence of operations using weighted unions but **not** path compression.

3. Draw the final data structure that results from applying the above sequence of operations using both weighted unions and path compression.
Minimum Spanning Trees (Chapt 24)

**Definition:** Given an undirected graph $G = (V, E)$ with weights on the edges, a **minimum spanning tree** of $G$ is a subset $T \subseteq E$ such that $T$ has

- no cycles
- contains all nodes in $V$
- sum of the weights of all edges in $T$ is minimum
Kruskal’s MST Algorithm

Idea:
• use a **greedy strategy**
• consider edges in increasing order of weight
• add edge to \( T \) if it doesn’t create a cycle

\[
\text{MST-KRUSKAL}(G) \\
1. \ T := \emptyset \\
2. \textbf{for each} \ v \in V \\
\quad \text{create}(v) /\textbf{**make singleton set**}/ \\
3. \text{sort edges in} \ E \ \text{by increasing weight} \\
4. \textbf{for each} \ (u, v) \in E \ /\textbf{**in sorted order**}/ \\
\quad \text{if} \ \text{find}(u) \neq \text{find}(v) \ /\textbf{**doesn’t create a cycle**}/ \\
\quad \quad \ T := T \cup \{(u, v)\} /\textbf{**add edge to MST**}/ \\
\quad \quad \ \text{union}(\text{find}(u), \text{find}(v)) /\textbf{**put u and v in same CC**}/ \\
\textbf{endfor} \\
5. \textbf{return} \ T
\]

Running Time
• initialization \(-O(1) + O(n) + O(m \log m) = O(n + m \log m)\)
• \(2m\) iterations (two for each edge)
  - \(4m\) finds \(-O(m \log^* m)\) time
  - \(O(n)\) unions \(-O(n)\) time (at most \(n - 1\) unions)
• total: \(O(n + m \log m)\) time
  - note \(\log m = O(\log n)\) since \(m = O(n^2)\) and \(\log m = O(2 \log n)\)

Note: We only get this bound because of amortized analysis
Exercise: Kruskal’s MST Algorithm

\[
\text{MST-Kruskal}(G)
\]
1. \( T := \emptyset \)
2. \textbf{for each } \( v \in V \)
   \begin{align*}
   &\text{create}(v) \quad /\!*\text{make singleton set}\!*/ \\
   &3. \text{sort edges in } E \text{ by increasing weight}
   \end{align*}
4. \textbf{for each } \( (u, v) \in E \ /\!*\text{in sorted order}\!*/ \\
   \begin{align*}
   &\text{if } \text{find}(u) \neq \text{find}(v) \ /\!*\text{doesn't create a cycle}\!*/ \\
   &\quad T := T \cup \{(u, v)\} \ /\!*\text{add edge to MST}\!*/ \\
   &\quad \text{union}(\text{find}(u), \text{find}(v)) \ /\!*\text{put } u \text{ and } v \text{ in same CC}\!*/
   \end{align*}
   \textbf{endfor}
5. \textbf{return } T

\begin{center}
\begin{tikzpicture}
\node[shape=circle,draw=black] (a) at (0, 2) {a};
\node[shape=circle,draw=black] (b) at (2, 3) {b};
\node[shape=circle,draw=black] (c) at (4, 2) {c};
\node[shape=circle,draw=black] (d) at (1, 0) {d};
\node[shape=circle,draw=black] (e) at (3, 0) {e};
\node[shape=circle,draw=black] (f) at (0, -2) {f};
\node[shape=circle,draw=black] (g) at (2, -2) {g};
\node[shape=circle,draw=black] (h) at (4, -2) {h};

\draw (a) -- (b) node [midway, above] {1};
\draw (b) -- (c) node [midway, above] {6};
\draw (a) -- (d) node [midway, right] {6};
\draw (d) -- (e) node [midway, above] {2};
\draw (e) -- (c) node [midway, right] {3};
\draw (e) -- (f) node [midway, above] {5};
\draw (e) -- (g) node [midway, above] {2};
\draw (g) -- (h) node [midway, above] {7};
\end{tikzpicture}
\end{center}

Exercise:

1. Show the workings of Kruskal’s MST algorithm on the graph shown. In particular, number the edges in the order in which they are added to the MST.
Correctness of Kruskal’s Algorithm

Theorem: Kruskal’s MST algorithm produces a MST.

Proof: Clearly algorithm produces a spanning tree. We need to argue it is an MST.

Suppose in contradiction it is not an MST. Suppose that the algorithm adds edges to the tree in order $e_1, e_2, \ldots, e_{n-1}$ and let $i$ be the value such that $e_1, e_2, \ldots, e_{i-1}$ is a subset of some MST $T$, but $e_1, e_2, \ldots, e_{i-1}, e_i$ is not a subset of any MST.

Consider $T \cup \{e_i\}$

- $T \cup \{e_i\}$ must have a cycle $c$ involving $e_i$
- in the cycle $c$ there is at least one edge that is not in $e_1, e_2, \ldots, e_{i-1}$ (since algorithm doesn’t pick $e_i$ if it makes a cycle)
- let $e^*$ be edge with minimum weight on $c$ that is not in $e_1, e_2, \ldots, e_i$
  - $wt(e_i) < wt(e^*)$ (else algorithm would have picked $e^*$)

Claim: $T' = T - \{e^*\} \cup \{e_i\}$ is a MST

- $T'$ is a spanning tree
  - contains all nodes
  - contains no cycles
- $wt(T') < wt(T)$, so $T$ is not MST (contradiction)

This contradiction means our original assumption must be wrong (so the algorithm does find an MST).

$\Box$
Prim’s MST Algorithm

Idea:

- always maintain a connected subgraph (different from Kruskal’s algorithm)
- at each iteration choose the cheapest edge that goes out from the current tree (a greedy strategy)

\[
\text{MST-PRIM}(G) \\
1. \quad v_0 := \text{any node in } V \\
2. \quad T := \emptyset \\
3. \quad \text{while } |T| < n \quad \\
\quad \quad E_{out} := \{(u, v) \in E | u \text{ in } T \text{ and } v \text{ not in } T\} \\
\quad \quad \text{let } (x, y) \text{ be edge in } E_{out} \text{ with minimum weight} \\
\quad \quad T := T \cup \{(x, y)\} \\
\quad \text{endwhile} \\
4. \quad \text{return } T
\]

Running Time – Adjacency List Rep

- \(n - 1\) iterations
- at each iteration, check all edges for minimum element of \(E_{out}\)
- total: \(O(nm)\) time

Note: we don’t find the minimum element of \(E_{out}\) very efficiently...
Exercise: Prim’s MST Algorithm

MST-PRIM(G)
1. \( v_0 := \text{any node in } V \)
2. \( T := \emptyset \)
3. while \( |T| < n - 1 \)
   \( E_{out} := \{(u, v) \in E | u \text{ in } T \text{ and } v \text{ not in } T\} \)
   let \((x, y)\) be edge in \(E_{out}\) with minimum weight
   \( T := T \cup \{(x, y)\}\)
endwhile
4. return \( T \)

Exercise:
1. Show the workings of Prim’s MST algorithm on the graph shown. In particular, number the edges in the order in which they are added to the MST. Assume the algorithm starts with the node labeled \( e \), i.e., \( v_0 = e \).
Prim’s MST Algorithm – more efficient

**Idea:** Use a priority queue – associate with each node \( v \) two fields:

- \( \textbf{best-wt}[v] \): if \( v \) isn’t in \( T \), then holds the min wt of all edges from \( v \) to a node in \( T \) (initially \( \infty \))
- \( \textbf{best-node}[v] \): if \( v \) isn’t in \( T \), then holds the name of the node in \( T \) such that \( wt(u, v) \) is \( v \)'s best out

**MST-PRIM(\( G \))**
1. insert each \( v \in V \) into priority queue \( Q \) with key \( \text{(best-wt)} \) \( \infty \)
2. \( T := \emptyset \)
3. \( v_0 := \) any node in \( V \)
4. decrease-key\( (v_0, 0) \)
5. while \( Q \neq \emptyset \)
   1. \( u := \) extract-min\( (Q) \)
   2. if \( u \neq v_0 \)
      1. then \( T := T \cup \{(u, \text{best-node}[u])\} \)
   3. for each neighbor \( v \) of \( u \)
      1. if \( v \in Q \) and \( wt(u, v) < \text{best-wt}(v) \)
         1. \( \text{best-node}[v] := u \)
         2. \text{decrease-key}(v, \text{wt}(u, v))
      4. endfor
   5. endwhile
Prim’s MST Algorithm – Running Time

- Assume $Q$ is implemented with a binary heap (heapsort)
- How to tell if $v \in Q$ without searching heap?
  - keep array for nodes with boolean flag indicating if in heap

Now, for analyzing the algorithm:

- initialize $Q$: $O(n \log n)$ time (really faster since all have key $\infty$)
- decrease $v_0$’s key – $O(\log n)$ time
- while loop...

  in each of $n$ iterations of while loop: $O(n \log n)$
  - extract min – $O(\log n)$ time
  - update $T$ – $O(1)$ time

over all iterations (combined): $O(m \log n)$
  - check neighbors of $u$: $O(m)$ executions
    - if condition test and updating best node – $O(1)$ time
    - decreasing $v$’s key – $O(\log n)$ time

So, the grand total is:

$$O(n \log n + m \log n) = O((n + m) \log n)$$

which is asymptotically the same as Kruskal’s algorithm.

Note: Can use Fibonacci Heaps to implement $Q$ and get complexity of $O(m + n \log n)$ which is better if $n < m$
Prim’s MST Algorithm – Correctness

Let $T_i$ be the tree after the $i$th iteration of the while loop.

**Lemma:** For all $i$, $T_i$ is a subtree of some MST of $G$

**Proof:** by induction on $i$

* Basis: $i = 0$, $T_0 = \emptyset$, ok

* Inductive Hypothesis: Assume $T_i$ is a subtree of some MST $M$

* Induction Step: Now show that $T_{i+1}$ is a subtree of some MST (possibly different from $M$)

Let $(u,v)$ be the edge added in iteration $i + 1$

**case 1:** $(u,v)$ is an edge of $M$

  - clearly, $T_{i+1}$ is a subtree of $M$ (ok)

**case 2:** $(u,v)$ is not an edge of $M$

- there is a path $P$ in $M$ from $u$ to $v$ (because $M$ is a spanning tree)
- let $(x,y)$ be the first edge in $P$ with $x$ in $T_i$ and $y$ not in $T_i$

- $M' = M - \{(x,y)\} \cup \{(u,v)\}$ is another spanning tree
- Now we note that
  \[
  wt(M') = wt(M) - wt(x,y) + wt(u,v) \leq wt(M)
  \]
  since $(u,v)$ is the minimum weight outgoing edge from $T_i$
- Therefore, $M'$ is also a MST of $G$ and $T_{i+1}$ is a subtree of it.

\[\square\]
Single Source Shortest Paths (Chapt 25)

The Single Source Shortest Path Problem (SSSP)

- **input:** directed graph $G = (V, E)$ with edge weights, and a specific source node $s$
- **goal:** find a minimum weight path from $s$ to every other node in $V$ (sometimes we only want the cost of these paths)

![Diagram](image)

Applications:

- weights can be distances, times, costs, etc.

**Note:** BFS finds shortest paths for special case when all edge weights are 1
- the result of SSSP algorithm can be viewed as a tree rooted at $s$ containing a shortest (wrt to weights) path from $s$ to all other nodes
Negative Cycles in SSSP!!

**Warning!!:** negative weight cycles are a problem...

What is the shortest path from $a$ to $d$?

- path $a$-$c$-$e$-$d$:
  weight $-12 + 3 - 2 = -11$

- path $a$-$c$-$e$-$d$-$a$-$c$-$e$-$d$:
  weight $-12 + 3 - 2 + (10 - 12 + 3 - 2) = -12$

- path $a$-$c$-$e$-$d$-$a$-$c$-$e$-$d$-$a$-$c$-$e$-$d$:
  weight $-12 + 3 - 2 + (10 - 12 + 3 - 2) + (10 - 12 + 3 - 2) = -13$

- so... if we keep going around the cycle (d-$a$-$c$-$e$-$d$) we keep decreasing the weight of the path... and so the shortest path has weight $-\infty$

**Note:** To avoid such situations we require that the graph has no negative weight cycles
Dijkstra’s SSSP Algorithm

\[ \text{SSSP-DIJKSTRA}(G, s) \]
1. \( d[s] := 0 \) and insert \( s \) into \( Q \) with key \( d[s] \)
2. \( Q := \emptyset /*\text{priority queue}*/ \)
3. \textbf{for each} \( v \neq s \)
   \( d[v] := \infty \)
   insert \( v \) into \( Q \) with key \( d[v] \)
4. \textbf{while} \( Q 
eq \emptyset \)
   \( u := \text{extract-min}(Q) \)
   \textbf{for each} neighbor \( v \) of \( u \)
   \( d[v] := \min\{d[v], d[u] + wt(u, v)\} \) (decrease-key(\( v \)) if min changes)
   /* \( d[v] \) is min \( s\to v \) path using nodes in \( S=V-Q* */
\textbf{endfor}
\textbf{endwhile}

\textbf{Note:} Dijkstra’s Algorithm only works if no negative weight edges

\textbf{Example:}

| iter | \( u \) | \begin{align*} \text{keys (of elts still in } Q) \\
\text{d[a]} & \text{d[b]} & \text{d[c]} & \text{d[d]} & \text{d[e]} \end{align*} |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>1</td>
<td>( a )</td>
<td>x</td>
<td>2</td>
<td>12</td>
<td>( \infty )</td>
</tr>
<tr>
<td>2</td>
<td>( b )</td>
<td>x</td>
<td>x</td>
<td>10</td>
<td>( \infty )</td>
</tr>
<tr>
<td>3</td>
<td>( c )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>( e )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>( d )</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Exercise: Dijkstra’s SSPP Algorithm

SSPP-\textsc{Dijkstra}(G, s)
1. \(d[s] := 0\) and insert \(s\) into \(Q\)
2. \(Q := \emptyset \) /*priority queue*/
3. for each \(v \neq s\)
   \hspace{1em} \(d[v] := \infty\)
   \hspace{1em} insert \(v\) into \(Q\) with key \(d[v]\)
4. while \(Q \neq \emptyset\)
   \hspace{1em} \(u := \text{extract-min}(Q)\)
   \hspace{1em} for each neighbor \(v\) of \(u\)
   \hspace{2em} \(d[v] := \min\{d[v], d[u] + wt(u, v)\}\) (decrease-key\((v)\) if min changes)
   \hspace{2em} /* \(d[v]\) is min \(s \rightarrow v\) path using nodes in \(S=V-Q\)*/
   \hspace{2em} endfor
   endwhile

Exercises:

1. Give a simple example of a directed graph with negative-weight edges (but no negative weight cycles) for which Dijkstra’s algorithm produces incorrect answers.

2. Suppose we change line 4 of Dijkstra’s algorithm to the following:
   \hspace{1em} \textbf{while } |Q| > 1
   This change causes the \textbf{while} loop to execute \(n - 1\) times instead of \(n\) times. Is this proposed algorithm correct? Why or why not?
Dijkstra’s SSSP Algorithm – Running Time

SSSP-DIJKSTRA($G, s$)
1. $d[s] := 0$ and insert $s$ into $Q$
2. $Q := \emptyset$ /*priority queue*/
3. for each $v \neq s$
   \begin{align*}
   d[v] &:= \infty \\
   \text{insert } v \text{ into } Q \text{ with key } d[v]
   \end{align*}
4. while $Q \neq \emptyset$
   \begin{align*}
   u &:= \text{extract-min}(Q) \\
   \text{for each neighbor } v \text{ of } u \\
   d[v] &:= \min \{d[v], d[u] + wt(u, v)\} \text{ (decrease-key($v$) if min changes)} \\
   /* d[v] is min s-$\rightarrow$v path using nodes in S=V-Q*/
   \end{align*}
endfor
endwhile

Running Time: (assume adjacency list representation)
The time will depend on implementation of priority queue $Q$...

in general:

- initialization (steps 1-3) – $O(n)$ time
- while loop has $n$ iterations
  - suppose extract-min takes $O(X_Q)$ time
- for loop executed $m$ times overall
  - suppose decrease-key takes $O(Y_Q)$ time
- total: $O(nX_Q + mY_Q)$ time
Dijkstra’s SSSP Algorithm – Running Time

in general $O(nX_Q + mY_Q)$ time

- extract-min takes $O(X_Q)$ time
- decrease-key takes $O(Y_Q)$ time

if $G$ is dense (i.e., $\Theta(n^2)$ edges):

- no point in being clever about extract-min, use matrix for $Q$ (size $n$, index by $v$)
- $X_Q = n$ and $Y_Q = \Theta(1)$
- total: $O(n^2 + m) = O(n^2)$ time

if $G$ is sparse (i.e., $o(n^2)$ edges):

- try to minimize extract-min, use binary heap (heapsort)
- $X_Q = O(\log n)$ and $Y_Q = O(\log n)$
- total: $O(n \log n + m \log n) = O((n + m) \log n)$ time
- (Fibonacci heaps give $O(n \log n + m)$ since $Y_Q = O(1)$ )
Dijkstra’s SSSP Algorithm – Correctness

**Lemma:** Recall $S = V - Q$, the nodes that have been removed from $Q$. For all nodes $x \in V$:

(a) if $x \in S$ ($x \not\in Q$), then the shortest $s$ to $x$ path only uses nodes in $S$ and $d[x]$ is its weight.

(b) if $x \not\in S$ ($x \in Q$), then $d[x]$ is weight of the shortest $s$ to $x$ path all of whose intermediate nodes are in $S$.

**Proof:** By induction on $i$, the number of iterations of the while loop.

**Basis:** $i = 1$, $S = \{s\}$, and $d[x] = \infty$ if $x$ is not a neighbor of $s$, and otherwise $d[x] = wt(s, x)$ (the edge weight).

So both (a) and (b) hold.

**Inductive Hypothesis:** Assume the Lemma is true for iteration $i - 1$.

**Induction Step:** Show it is true for iteration $i$. Let $u$ be the node selected in the $i$th iteration (the node in $Q$ with minimum $d[u]$ value).

**Proof of (a)**

**case 1:** $x \neq u$. Then $x$ was in $S$ before iteration $i$, and by the inductive hypothesis, we already had the best $s \rightarrow x$ path.

**case 2:** $x = u$. Suppose in contradiction that the shortest $s \rightarrow x$ path uses some node $r$ not in $S$ after iteration $i$.

- $d[u]$ is wt of shortest $s \rightarrow u$ path with all intermediate nodes in $S$ (induct hyp)
- $d[r]$ is wt of shortest $s \rightarrow r$ path with all intermediate nodes in $S$ (induct hyp)
- $d[u] \leq d[r]$ since alg picks $u = x$

So... the shortest $s \rightarrow u$ path can’t go thru $r$ since there are no negative weights
Dijkstra’s SSSP Algorithm – Correctness

Proof of (b)
Choose $x$ that is not in $S$ after iteration $i$

- $d[x]$ is wt of shortest $s \rightarrow x$ path with all intermediate nodes in $S$ (induct hyp)
- $d[u]$ is wt of shortest $s \rightarrow u$ path (by (a))

**case 1:** $x$ is not a neighbor of $u$
The adding $u$ to $S$ doesn’t change the best $s \rightarrow x$ path with all internal nodes in $S$. To see why not, suppose it did:

- there is a better $s \rightarrow v$ path obtained by going outside $S$... This contradicts (a)!!!
- so best $s \rightarrow x$ path, and $d[x]$, can’t change

**case 2:** $x$ is a neighbor of $u$

- alg checks to see if it is better to go from $s$ to $x$ via $u$, or to stick with the original path
Exercise: Correctness of Dijkstra’s SSSP Algorithm

Lemma: Recall $S = V - Q$, the nodes that have been removed from $Q$. For all nodes $x \in V$:

(a) if $x \in S$ ($x \notin Q$), then the shortest $s$ to $x$ path only uses nodes in $S$ and $d[x]$ is its weight.

(b) if $x \notin S$ ($x \in Q$), then $d[x]$ is weight of the shortest $s$ to $x$ path all of whose \textit{intermediate} nodes are in $S$

Exercise:

1. Why doesn’t the proof of correctness for Dijkstra’s algorithm go through when negative-weight edges are allowed?
Dynamic Programming Overview

Dynamic Programming – What and When?

• some recursive divide-and-conquer algorithms are inefficient – they solve the same subproblem more than once...

• dynamic programming is a technique that can be used to cut down on this inefficiency (solve each subproblem just once)

• typically applied to **optimization problems**

There are two basic approaches to dynamic programming...

Method 1: Top-Down Recursive Approach (Memorization in text)

• start with recursive divide-and-conquer algorithm

• keep ‘top-down’ approach of original algorithm (easy)

• save solutions to subproblems in a table (potentially a lot of storage)

• when running the algorithm, only recurse on a subproblem if the solution is not already available in the table

Method 2: Bottom-Up Iterative Approach

• start with recursive divide-and-conquer algorithm

• study the problem and algorithm and figure out the dependencies between the subproblems (which solutions are needed for each subproblem)

• rewrite the algorithm so it solves the subproblems in the correct order (so won’t have to save as many solutions in the table and it can be smaller)
Review: The Divide-And-Conquer Paradigm

The divide-and-conquer paradigm is one of the most useful and common paradigms for designing algorithms. It solves a problem by breaking it into two or more subproblems:

1. divide problem into subproblems
2. solve subproblems (usually recursively)
3. combine subproblem solutions to get answer

Example: Mergesort

```mergesort(A)
{
    1. A1 = mergesort(first half of A)
    2. A2 = mergesort(second half of A)
    3. merge(A1, A2)
}
```

Running Time:

- Step 1: $T(n/2)$
- Step 2: $T(n/2)$
- Step 3: $O(n)$

so.... we get the following recurrence relation for the running time:

$$T(n) = 2T(n/2) + O(n)$$

$$= O(n \log n)$$
Dynamic Programming Ex: All Pairs Shortest Paths (Chapt 26)

The All Pairs Shortest Path Problem (APSP)

- **input:** directed graph \( G = (V, E) \) with edge weights
- **goal:** find a minimum weight path between every pair of vertices in \( V \) (sometimes we only want the cost of these paths)

![Graph Diagram](image)

Solution 1: run Dijkstra’s algorithm \( n \) times, once with each \( v \in V \) as the source node (no negative weight edges in \( E \))

- \( G \) is dense – array implementation of \( Q \)
  - \( O(n \cdot (n^2)) = O(n^3) \) time
- \( G \) is sparse – binary heap implementation of \( Q \)
  - \( O(n \cdot ((n + m) \log n)) = O(n^2 \log n + nm \log n) \) time

Solution 2: Floyd-Warshall Algorithm

- introduces **dynamic programming** technique
Floyd-Warshall APSP Algorithm

Use adjacency matrix $A$ for $G = (V, E)$:

$$A[i, j] = a_{ij} = \begin{cases} w(i, j) & \text{if } (i, j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \text{ and } (i, j) \notin E \end{cases}$$

**Goal:** Compute $n \times n$ matrix $D$ where:

$$D[i, j] = d_{ij} = \text{weight of shortest } i \text{ to } j \text{ path}$$

**Definition:** Let $D^{(k)}$ be a $n \times n$ matrix where:

- $D^{(k)}[i, j] = d^{(k)}_{ij} = \text{weight of shortest } i \text{ to } j \text{ path all of whose intermediate nodes are in } \{1, 2, \ldots, k\}$
- $D^{(0)} = A$, original adjacency matrix (only paths are single edges)
- $D^{(n)}$ is the matrix we want to compute
- $D^{(k)}$’s elements are:

$$D^{(k)}[i, j] = d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$$

Only intermediate nodes in paths are in $(1, \ldots, k-1)$
Floyd-Warshall APSP Algorithm – Recursive

This definition leads to a nice recursive algorithm to compute $D^{(n)}$:

$$D^{(k)}[i, j] = d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$$

**APSP-RECURSIVE($G$)**

1. for $i := 1, n$
2. for $j := 1, n$
   
   $D^{(n)}[i, j] := \text{rsp}(i, j, n)$  /**shortest i->j path**/

**RSP($i, j, k$)  /* computes $D^{(k)}[i, j]$ */**

1. if $(k = 0)$
   
   then return $A[i, j]$  /**no intermediate nodes**/

   else return $\min(\text{rsp}(i, j, k - 1), \text{rsp}(i, k, k - 1) + \text{rsp}(k, j, k - 1))$

**Running Time:** exponential!! – $\approx 3^n$

**why?** lots of repeated subproblems...

**Example:** for $n = 3$ and solving for $D^{(3)}[1, 2]$
Floyd-Warshall APSP – ‘Top-Down’ Dynamic Programming

Dynamic Programming Idea: solve each subproblem only once, save the solution and look it up when needed (don’t resolve)

APSP-dynamic(G)
1. for i := 1, n
2. for j := 1, n
3. for k := 0, n
4. \( D^{(k)}[i,j] := \text{NIL} /*\text{initialize}*/ 
5. for i := 1, n
6. for j := 1, n
7. \( D^{(n)}[i,j] := \text{dsp}(i,j,n) /*\text{shortest i->j path}*/ 

DSP(i,j,k)
1. if \( D^{(k)}[i,j] = \text{NIL} /*\text{if didn’t already compute}*/ 
2. if k = 0
3. then \( D^{(k)}[i,j] = A[i,j] /*\text{no intermediate nodes}*/ 
4. else \( D^{(k)}[i,j] = \text{min}(\text{dsp}(i,j,k-1),\text{dsp}(i,k,k-1)+\text{dsp}(k,j,k-1)) 
5. endif
6. return \( D^{(k)}[i,j] 

Running Time: \( O(n^3) \) now!
why? there are \( O(n^3) \) distinct subproblems (which ones?)
and each one is ‘solved’ only once (if NIL)
Floyd-Warshall APSP – ‘Bottom-Up’ Dynamic Programming

**Problem:** The storage required to store all the subproblem solutions is $O(n^3)$ (or $O(n^3)$) since we have $n + 1$ matrices, each of size $n \times n = n^2$.

**Idea:** Actually, we can do much better (only $O(n^2)$ space using only one $D$ array) by solving the subproblems in the 'right' order...

- solve 'smallest' first (i.e., $k = 1, 2, \ldots$)

**APSP-FLOYD-WARSHALL($G$)**
1. $D := A$ /*initialize to adj matrix*/
2. for $k := 1, n$
3. for $i := 1, n$
4. for $j := 1, n$

$$D[i, j] := \min(D[i, j], D[i, k] + D[k, j])$$

**Example:**

![Graph Image](image-url)

![Matrix Image](matrix-url)
Transitive Closure (modified Floyd-Warshall APSP)

The Transitive Closure Problem:

- **input:** a directed graph \( G = (V, E) \) (weighted or not)
- **output:** an \( n \times n \) matrix \( T \) such that \( T[i, j] = 1 \) iff there is a path from \( i \) to \( j \) in \( G \)

**Algorithm:**

- Assign a weight of 1 to each edge in \( E \)
- Run Floyd-Warshall Algorithm
- Fill in \( T \) array so that \( T[i, j] = 1 \) iff \( D^{(n)}[i, j] \neq \infty \)
- runs in \( O(n^3) \) time and uses \( O(n^2) \) space (just like Floyd-Warshall algorithm)
- Note: can be a bit smarter and instead of computing length of shortest paths, just note whether one exists (can use Boolean \texttt{and/or} operations instead of \texttt{min} of path lengths)
Dynamic Programming Example: Matrix Multiplication Order

Recall: if $A$ is $m \times n$ and $B$ is $n \times p$, then $A \cdot B = C$ is $m \times p$ and

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]$$

and the time need to compute $C$ is $O(mnp)$

- there are $mp$ elements of $C$
- each one requires $n$ scalar multiplications and $n-1$ scalar additions
  - $O(n)$ time

Problem: Given matrices $A_1, A_2, \ldots, A_n$, where the dimension of $A_i$ is $d_{i-1} \times d_i$, determine the minimum number of multiplications needed to compute the product $A_1 \cdot A_2 \cdot \ldots \cdot A_n$.

Two possibilities:

- $(A_1 \cdot A_2) \cdot A_3$ – total multis $60 = 40 + 20$
  - $M_1 := A_1 \cdot A_2$: requires $40 = 4 \cdot 2 \cdot 5$ multis, $M_1$ is $4 \times 5$ matrix
  - $M_1 \cdot A_3$: requires $20 = 4 \cdot 5 \cdot 1$ multis

- $A_1 \cdot (A_2 \cdot A_3)$ – total multis $18 = 10 + 8$
  - $M_1 := A_2 \cdot A_3$: requires $10 = 2 \cdot 5 \cdot 1$ multis, $M_1$ is $2 \times 1$ matrix
  - $A_1 \cdot M_1$: requires $8 = 4 \cdot 2 \cdot 1$ multis

So the order of multiplication can make a big difference!
Matrix Multiplication Order – Recursive Solution

Recursive Solution
• let $M[i, j] = \min$ number mults to compute $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$
  - note, dimension of $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ is $d_{i-1} \times d_j$
  - $M[i, i] = 0$ for $i = 1, n$
  - $M[1, n]$ is the solution we want
• $M[i, j]$ can be determined as follows:
  $$M[i, j] = \min_{i \leq k < j} (M[i, k] + M[k+1, j] + d_{i-1}d_kd_j)$$

Can we afford to check all possible orders?
• let $P(r)$ denote the number of ways to multiply $r$ matrices, $P(1) = 1$
• for $r > 1$, if we pick a multiplication to do last
  $$(A_1 \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_n)$$
  there are $P(k) \cdot P(n - k)$ ways to do the remaining multiplications
• there are $n - 1$ choices for the last multiplication... so
  $$P(n) = \sum_{k=1}^{n-1} P(k) \cdot P(n - k)$$
• the solution to this recurrence is $\Omega(4^n/n^{3/2})$, which is exponential in $n$, the number of matrices.
• so let's try dynamic programming!

- use an $n \times n$ matrix (table) $M[]$ to store subproblem solutions
  - $M[i, j]$ stores the min mults for $A_i \cdot \ldots \cdot A_j$ (above diagonal only)
- initialize: $M[i, j] := \text{NIL}$ for all $1 \leq i < j \leq n$, and $M[i, i] := 0$
- answer is $M[1, n]$

\[
\text{MM}(i, j)
\]
1. if $M[i, j] = \text{NIL}$
2. $M[i, j] := \infty$
3. for $k = i, j - 1$
4. $M[i, j] := \min(M[i, j], \text{MM}(i, k) + \text{MM}(k + 1, j) + d_{i-1} d_k d_j)$
5. endfor
5. endif
6. return $M[i, j]$

---

Running Time and Space
- $O(n^2)$ space for $M[]$ matrix (table)
- $O(n^3)$ time to fill in $O(n^2)$ entries
  - each one takes $O(n)$ time
Matrix Mult. Order: ‘Bottom-Up’ Dynamic Programming

- note that when computing $M[i, j]$ we only need the entries that are in the diagonals to the left of $M[i, j]$ (solve subproblems in order of diagonal)

```
MMORDER(1, n)
1. for i := 1, n
2. $M[i, i] = 0$
3. for d := 1, n - 1 /**diagonals**/
4. for i := 1, n - d /**rows**/
5. j := i + d /**col for row i entry on diag d**/
6. $M[i, j] := \infty$
7. for k = i, j - 1
8. $M[i, j] := \min(M[i, j], M[i, k] + M[k + 1, j] + d_{i-1}d_kd_j)$
9. return $M[i, j]$
```

Running Time and Space
- $O(n^2)$ space for $M[]$ matrix (table) (the same)
- $O(n^3)$ time to fill in $O(n^2)$ entries (the same)
  - each one takes $O(n)$ time
Dynamic Programming Example: Longest Common Subsequence

Problem: Given $X = < x_1, x_2, \ldots, x_m >$ and $Y = < y_1, y_2, \ldots, y_n >$, find the longest common subsequence (LCS) of $X$ and $Y$.

Example:
- $X = < A, B, C, B, D, A, B >$
- $Y = < B, D, C, A, B, A >$
- $LCS = < B, C, B, A >$ (or also $LCS = < B, D, A, B >$)

Brute-Force Solution:
- enumerate all subsequences of $X$ and check to see if they appear in $Y$.
- each subsequence of $X$ corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of the elements of $X$
  - so there are $2^m$ subsequences of $X$ (why?)
- clearly, this is not a good approach.... time for dynamic programming!
LCS – Recursive Sol and ‘Top-Down’ Dynamic Programming

The Recursive LCS Formulation

- let $C[i, j] = \text{the length of the LCS of } X_i \text{ and } Y_j$, where
  - $X_i = <x_1, x_2, \ldots, x_i>$ and $Y_j = <y_1, y_2, \ldots, y_j>$.
- our goal: $C[m, n]$ (consider entire $X$ and $Y$)
- basis: $C[0, j] = 0$, and $C[i, 0] = 0$
- $C[i, j]$ is calculated as shown below (two cases): 

  **Case 1: $x_i = y_j \ (i, j > 0)$**
  In this case, we can increase the size of the LCS of $X_{i-1}$ and $Y_{j-1}$ by one by appending $x_i = y_j$ to the LCS of $X_{i-1}$ and $Y_{j-1}$, i.e.,
  $$C[i, j] = C[i - 1, j - 1] + 1$$

  **Case 2: $x_i \neq y_j \ (i, j > 0)$**
  In this case, we take the LCS to be the longer of the LCS of $X_{i-1}$ and $Y_j$, and the LCS of $X_i$ and $Y_{j-1}$, i.e.,
  $$C[i, j] = \max(C[i, j - 1], C[i - 1, j])$$

The ‘Top-Down’ Dynamic Programming Solution

- initialize $C[i, 0] = C[0, j] = 0$ for $i = 0, m$ and $j = 1, n$
- initialize $C[i, j] = \text{NIL}$ for $i = 1, m$ and $j = 1, n$.

  LCS$(i, j)$
  1. if $C[i, j] = \text{NIL}$
  2. if $x_i = y_j$
  3. then $C[i, j] = \text{lcs}(i - 1, j - 1) + 1$
  4. else $C[i, j] = \max(\text{lcs}(i, j - 1), \text{lcs}(i - 1, j))$
  7. return $C[i, j]$
LCS – ‘Bottom-Up’ Dynamic Programming

We now want to figure out the ‘right’ order to solve the subproblems...

To compute $C[i, j]$ we need the solutions to:

- $C[i - 1, j - 1]$ (when $x_i = y_j$)
- $C[i - 1, j]$ and $C[i, j - 1]$ (when $x_i \neq y_j$)

if we fill in the $C$ array in row major order, these dependences will be satisfied.

1. for $j = 0, n$ $C[0, j] = 0$ (init)
2. for $i = 0, m$ $C[i, 0] = 0$ (init)
3. for $i = 1, m$
4. for $j = 1, n$
   - if $x_i = y_j$
   - then $C[i, j] = C[i - 1, j - 1] + 1$
   - else $C[i, j] = \max(C[i, j - 1], C[i - 1, j])$
7. return $C[m, n]$

Running Time: $O(mn)$ (constant time for each entry)

How can we keep track of the sequence?
We need to keep track of which neighboring table entry gave the optimal solution

- if $x_i = y_j$, the answer came from the diagonal
- if $x_i \neq y_j$, the answer came from the left or the right

we can keep another table to keep this info, when we’re done just trace back
from $C[m, n] –$ takes $O(m + n)$ time to trace back to square $C[1, 1]$

**Practice Problem:** show $C$ for $X =< a, b, b >$ and $Y =< b, a, b >$