Dynamic programming (part 1)

CS 146 - Spring 2017

Today

- Last time:
  - Bellman-For algorithm
  - Shortest paths with negative weights
- Floyd-Warshall algorithm
- main ideas behind dynamic programming

idea: relax ALL edges, V -1 times

```
bellmanFord(G, s) {
    Map dist = new Map();
    for each vertex v of G
        dist.put(v , infinity)
    dist.put(s, 0)

    repeat V - 1 times
        for each vertex v of G
            for neighbor w of v
                relax(v -> w)
    return dist;
}
```

initialize as before

one round of relaxation

time complexity: O(VE)

Observation:
if the shortest path tree has depth k,
there are no more changes to the distances after k rounds of relaxation

```
bellmanFord(G, s) {
    Map dist = new Map();
    for each vertex v of G
        dist.put(v , infinity)
    dist.put(s, 0)


    repeat V - 1 times
        changed = false
        for each vertex v of G
            for neighbor w of v
                if (dist(w) > dist(v) + w(v->w))
                    dist(w) = dist(v) + w(v->w)
                    changed = true
        if (!changed) break
    return dist;
}
```

Bellman-Ford with optimization

```
bellmanFord(G, s) {
    Map dist = new Map();
    for each vertex v of G
        dist.put(v , infinity)
    dist.put(s, 0)

    repeat V - 1 times
        for each vertex v of G
            for neighbor w of v
                relax(v -> w)

    for each vertex v of G
        for neighbor w of v
            if (dist(v) > dist(w) + w(v->w)
                error("negative cycle detected");
    return dist;
}
```

one more round of relaxation to see if there are any more changes

Bellman-Ford with cycle detection

In summary… single source shortest-path problem

- 2 relaxation-based algorithms
  - Dijkstra’s algorithm: O(E log V), edges non-neg
  - Bellman-Ford algorithm: O(VE), neg edges OK

All-pairs shortest paths

- input: a weighted graph (potentially with negative weights)
- output: a dictionary or matrix of distances between every pair of vertices

give an algorithm for this problem and its runtime

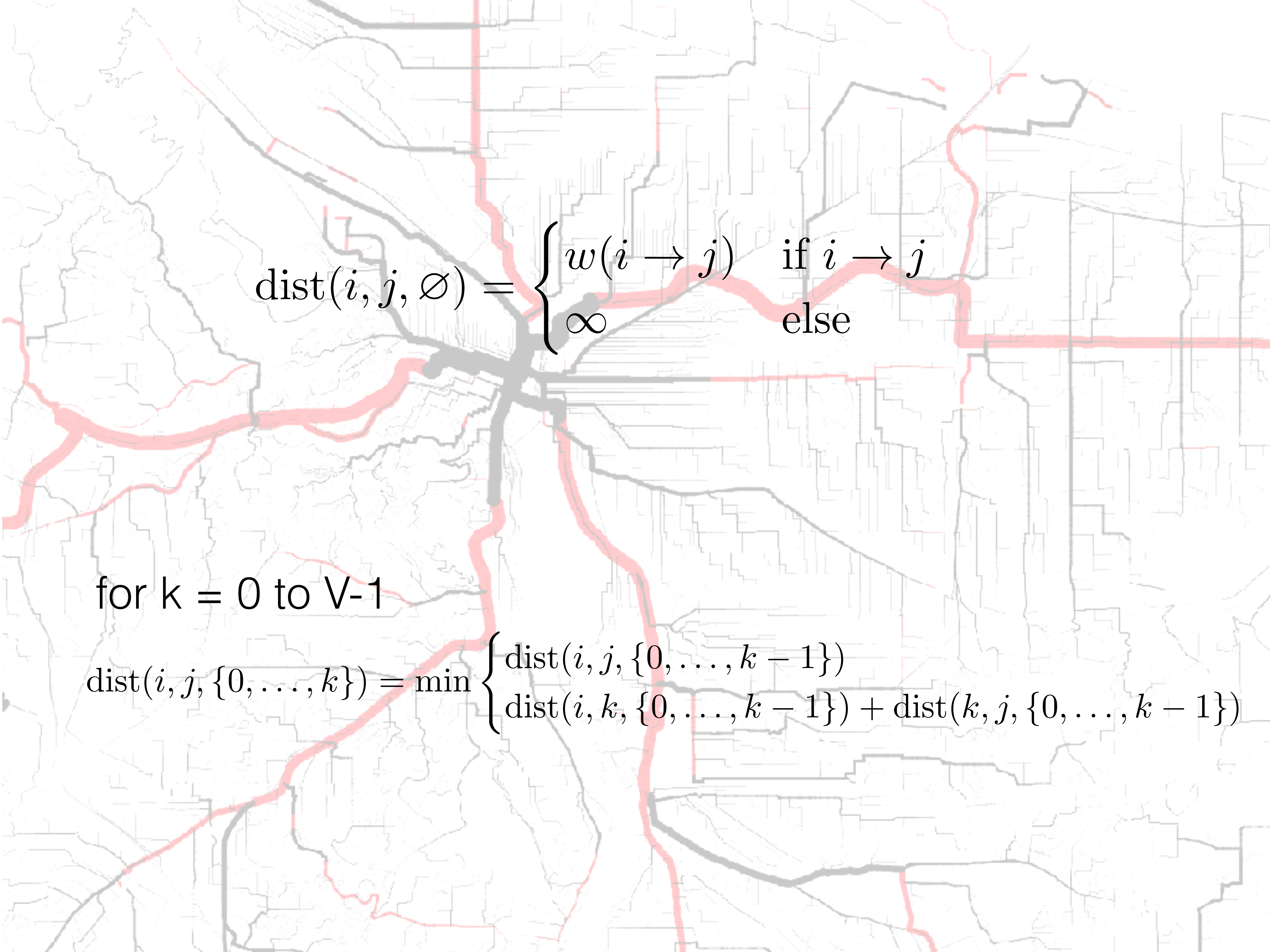
1st attempt: use Bellman-Ford multiple times

- for each vertex v
  - run Bellman-Ford with v as the source
  - record distances from v to every other vertex
- time: O(V^2 E)
- space: O(V^2)

seems like a lot of
work being redone, can
we do better?

2nd attempt: recursion + memoization

- record every shortest path between each pair as we go along number the vertices 0, 1, 2, 3, … V - 1
- subproblem: **dist(i, j, S) = distance from i to j using only vertices in set S to connect them**
- base case?
- problem in terms of subproblem?



$$\mathrm{dist}(i, j, \varnothing) = \begin{cases} w(i \to j) & \text{if } i \to j \\ \infty & \text{else} \end{cases}$$

for k = 0 to V-1

$$\mathrm{dist}(i, j, \{0, \ldots, k\}) = \min \begin{cases} \mathrm{dist}(i, j, \{0, \ldots, k-1\}) \\ \mathrm{dist}(i, k, \{0, \ldots, k-1\}) + \mathrm{dist}(k, j, \{0, \ldots, k-1\}) \end{cases}$$

Recall longest increasing subsequence

Given a sequence of numbers

for example: 2, 4, 3, 5, 1, 7, 6, 9, 8

What is the length of the longest increasing subsequence?

strictly

going left to right,
possibly skipping some

Making recursion more time efficient

(by using more space efficient)

	memoization (top-down)	dynamic programming (bottom up)
store subproblems in …	dictionary	array (2D if subpb has 2 params)
subproblem input	dictionary key	array index
subproblem output	dictionary value	entry at given index
code structure	recursive	for-loops

$$\mathrm{dist}(i, j, \varnothing) = \begin{cases} w(i \to j) & \text{if } i \to j \\ \infty & \text{else} \end{cases}$$

$$\mathrm{dist}(i, j, \{0, \ldots, k\}) = \min \begin{cases} \mathrm{dist}(i, j, \{0, \ldots, k-1\}) \\ \mathrm{dist}(i, k, \{0, \ldots, k-1\}) + \mathrm{dist}(k, j, \{0, \ldots, k-1\}) \end{cases}$$

for k = 0 to V-1

represented in 3d array

`dist[i][j][k] = distance from i to j using vertices 0 up to` **`k-1`**`, where k > 0`

(shift k by 1 to have room for empty set)

Floyd-Warshall algorithm for all-pairs shortest path algorithm

```
int[][][] dist = new int[V][V][V+1]

for (int i = 0; i < V; i++)

    for (int j = 0; j < V; j++)

        dist[i][j][0] = i -> j ? w(i -> j) : (i == j ? 0 : inf)

for (int k = 1; k < V+1; k++)

    for (int i = 0; i < V; i++)

        for (int j = 0; j < V; j++)

            dist[i][j][k] = min(dist[i][j][k-1],

                                dist[i][k][k-1] + dist[k][j][k-1])

return dist;
```

time and space complexity?

How to reconstruct the paths?

- record choices made at every step
- start at the beginning or the end (whichever is appropriate) and follow choices to reconstruct path
- same approach as with Dijkstra’s algorithm, making change problem, recursive problems etc.

record choices made at every step:
next[][s][V] stores shortest path tree **to** s
(s is the sink or destination, rather than source)

in other words…

```
next[u][v][V] = w
```

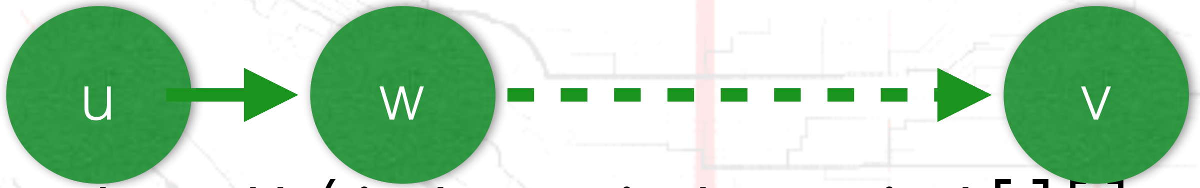
means

w comes right after u on
shortest path from u to v

```
int[][][] dist = new int[V][V][V+1]; int[][][] next = new int[V][V][V+1]
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        dist[i][j][0] = i -> j ? w(i -> j) : (i == j ? 0 : inf)
        next[i][j][k] = i -> j ? j : null;
for (int k = 1; k < V+1; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (dist[i][j][k-1] < dist[i][k][k-1] + dist[k][j][k-1]) {
                dist[i][j][k] = dist[i][j][k-1];
                next[i][j][k] = next[i][j][k-1];
            } else {
                dist[i][j][k] = dist[i][k][k-1] + dist[k][j][k-1];
                next[i][j][k] = next[i][k][k-1];
return dist;
```

record choices made at every step:
next[][s][V] stores shortest path tree **to** s
(s is the sink or destination, rather than source)

Floyd-Warshall algo with path reconstruction (1/2)



`next[u][v][V] = w`

```
reconstruct-path(int u, int v, int[][] next) {
    List<Integer> path = new ArrayList<>();
    path.add(u);
    for (int w = u;  next[w][v] != null;  w = next[w][v])
        path.add(next[w][v]);
    return path;
}
```

```
shortest-path-with-Floyd-Warshall(G, u, v) {
    int[][][] next = Floyd-Warshall(G);
    return reconstruct-path(u, v, next[][][V]);
}
```

sample usage
of reconstruct-path

(not efficient: if only need one path, use Dijkstra)

Floyd-Warshall algo with path reconstruction (2/2)

Making recursion more time efficient

(by using more space efficient)

	memoization (top-down)	dynamic programming (bottom up)
store subproblems in …	dictionary	array (2D if subpb has 2 params)
subproblem input	dictionary key	array index
subproblem output	dictionary value	entry at given index
code structure	recursive	for-loops

can be automated

?

(see reading)