Lecture 15
Path finding using dynamic programming

EECS-214

All points shortest-path

- Dijkstra's algorithm is cool, but what if you need to do **a lot** of route finding in the same graph?

Memoizing

- One thing you could do is to **save** the shortest paths as you compute them
  - E.g. in a **hash table**
- Then **reuse** the saved path if you have to solve the same problem again

- This is called **memoizing**
  - Or sometimes **caching** (although that usually means something a little different)

```
ShortestPath(a, b)
  if (a/b already in table)
    return table[a, b]
  else
    path = dijkstra(a, b)
    table[a, b] = path
    return path
```

Can we do better?

- This saves us recomputing the same path twice
- But we still have to run Dijkstra's algorithm $\boldsymbol{V^2}$ **times** to compute all paths
- That's $\boldsymbol{O\left(\left(V^3 + EV^2\right)\log V\right)}$ time, which is a lot if V is large

Shortest paths contain other shortest paths

- The **shortest path** from A to B
  - Has to **start with A**
  - Go through some (possibly empty) set of **intermediate nodes**
  - And **end with B**

- If C is some intermediate node on the shortest path, then the path is just:
  - The **shortest path from A to C**
  - Followed by the **shortest path from C to B**

Paths within paths

- So in finding the shortest path from A to B,
  - We’re **implicitly finding** the shortest paths from A to C and C to B
  - We’re **re-solving** the same problems repeatedly

- We’d like some way of **memoizing** this work so we don’t have to resolve the same problems

Restating what we already said

- **For any nodes** A, B, and C
- Either
  - The shortest path from A to B is
    - The **shortest path from A to C**
    - Followed by the **shortest path from C to B**
  - Or the shortest path from A to B **doesn't go through C**

And restating it again..

ShortestPath(A,B) =

- ShortestPathNotUsingC(A,C) followed by ShortestPathNotUsingC(C,B)
- Or ShortestPathNotUsingC(A,B)
- Whichever is shorter

(where “not using C” means “not using C as an intermediate node”)

Why do we care?

- We just described
  - The shortest path between two points
  - In terms of shortest paths **not using** some other node (C)

- We can **recurse**
  - Describe shortest paths not using C
  - In terms of shortest paths **not using C or D** (for some D)

Why do we care?

- We can recurse:
  - Describe shortest paths not using C
  - In terms of shortest paths not using C or D (for some D)
- And we **keep recursing**
  - Until we describe paths in terms of shortest paths **not using *any*** intermediate nodes
- They're just **edges** (easy to compute)

The Floyd-Warshall Algorithm

- Assume the **vertices are numbered**
- Define $\boldsymbol{D(i,j,k)}$ (i.e. "distance") to be
  - The **length** of the shortest path
    - From node i to node j
    - **Using only nodes $\boldsymbol{0}$ through $\boldsymbol{k}$**
- We **just compute the length** here
  - It's **easy to extend** the algorithm to recover the actual path

The Floyd-Warshall Algorithm

Then

$$D(i,j,k) = \begin{cases} \text{edgecost}(i,j), & k = 0 \\ \min\begin{bmatrix} D(i,j,k-1) \\ D(i,k,k-1) + D(k,j,k-1) \end{bmatrix}, & \text{otherwise} \end{cases}$$

- This might not look like an algorithm, but it's easy to turn into one

The Floyd-Warshall Algorithm
(bad version)

```
Distance(i, j)
  return D(i, j, V)    // V = number of nodes in graph = highest node number

D(i, j, k) {
  if (k=0) {
    if there's an edge between i and j
      return edgeCost[i, j]
    else
      return infinity
  } else {
    direct = D(i, j, k-1)
    indirect = D(i, k, k-1)+D(k, j, k-1)
    return min(direct, indirect)
  }
}
```

uhhh....

weren’t we supposed to trying to memoize this computation?

The Floyd-Warshall Algorithm
(bad memoized version)

```
float[] distances = new float[V, V, V];     // 3D array indexed by vertex number

D(i, j, k) {
  if distances[i, j, k] has an entry
    return distances[i, j, k]
  if (k=0) {
    if there's an edge between i and j
      answer = edgeCost[i, j]
    else
      answer = infinity
  } else {
    direct = D(i, j, k-1)
    indirect = D(i, k, k-1)+D(k, j, k-1)
    answer = min(direct, indirect)
  }
  distances[i, j, k] = answer
  return answer
}
```

uhhh....

weren’t we supposed to trying to compute all paths at once?

And now the cleverness…

- We **compute all** the different **D(i, j, k)** values
- But we **only care** about the ones where **k=V**
  - i.e. where k is the number of vertices
  - i.e. where we're allowed to use all the vertices

- **Once we compute all the values for k**, we **don't care** about the k-1 values

The real Floyd-Warshall algorithm

- Compute **D(i, j, 0)** for all i, j
- Compute **D(i, j, 1)** for all i, j
  - **Throw away** the D(i, j, 0) values
- Compute **D(i, j, 2)** for all i, j
  - **Throw away** the D(i, j, 1) values

…

- Compute **D(i, j, V)** for all i, j
  - **Throw everything else away**

The real Floyd-Warshall algorithm

```
ComputeAllDistances() {
  D = new V × V array initialized to ∞
  for each vertex v, D[v, v] = 0
  for each edge e=(u,v),  D[u,v] = weight(e)
  for each vertex k
    for each vertex j
      for each vertex i
        D[i, j] = min(D[i,j], D[i,k]+D[k,j])
  return D
}
```

The real Floyd-Warshall algorithm

```
ComputeAllDistances() {
  D = new V × V array initialized to ∞
  for each vertex v, D[v, v] = 0
  for each edge e=(u,v),  D[u,v] = weight(e)
  for each vertex k
    for each vertex j
      for each vertex i
        D[i, j] = min(D[i,j], D[i,k]+D[k,j])
  return D
}
```

Just $O\left(V^3\right)$!

Dynamic programming

- Dynamic programming is the technique of
  - **Optimizing** algorithms
  - By **avoiding re-solving** subproblems
  - By **storing and reusing** the results of subproblems

- Can be as simple as just **memozing a recursion**
  - Sometimes called **top-down** dynamic programming
  - Main problem $\longrightarrow$ subproblems

- But it can also involve cleverly rearranging the subproblems so it doesn't even look like a recursion anymore
  - Subproblems $\longrightarrow$ main problem(s)
    - Like Floyd-Warshall
  - Called **bottom-up** dynamic programming

Classic algorithm design techniques

- **Divide and conquer**
  - Solve problem using subproblems
  - Example: binary search
- **Dynamic programming**
  - Store and reuse solutions to subproblems
  - Example: Floyd-Warshall
- **Randomization**
  - Avoid unlikely worst-case behavior
  - Example: randomized quicksort
- **Amortized analysis** (next)
  - Provide good bounds on efficiency of sequences of calls
  - Even if individual calls might be slow
- Probabilistic methods
  - Very likely produce the right answer
- Greedy optimization
  - Globally optimal set of choices from locally optimal individual choices
- Approximation
  - Answers that are provably close to optimal