Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

March 13, 2023 (Lecture 14)

All Pairs Shortest Paths

All Pairs Shortest Paths

Now we consider the problem of finding shortest paths between all pairs of vertices in a graph. Suppose we are given a weighted, directed graph $G = (V, E)$ with a weight function w: E $\rightarrow$ R that maps edges to real-valued weights. We wish to find, for every pair of vertices $u, v \in V$, a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. We typically want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v. We can run Dijkstra's algorithm or Bellman-ford algorithm for each of the vertices, but we want to find more efficient algorithms.

All Pairs Shortest Paths

We will use an adjacency-matrix representation of a graph instead of adjacency-list representation. For convenience, we assume that the vertices are numbered 1, 2, . . . , |V|, and the matrix representation of the directed graph is $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{the weight of edge}(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

We allow negative-weight edges, but the input graph contains no negative-weight cycle.

The Floyd-Warshall Algorithm

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in previous section.

The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l , that is, any vertex in the set $\{v_2, \ldots, v_{l-1}\}$.

The Floyd-Warshall Algorithm

As before, we assume that the vertices of G are V = {1, 2, . . . , n}.

Let us consider a subset {1, 2, . . . , k} of vertices for some k. For any pair of vertices i, j ∈ V , consider all paths from i to j whose intermediate vertices are all drawn from {1, 2, . . . , k}, and let p be a minimum-weight path from among them. (Path p is simple.)

The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2, . . . , k − 1}. The relationship depends on whether or not k is an intermediate vertex of path p.

The Floyd-Warshall Algorithm

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2, . . . , k−1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2, . . . , k−1} is also a shortest path from i to j with all intermediate vertices in the set {1, 2, . . . , k}.
- If k is an intermediate vertex of path p, then we decompose p into i (p_1) $\rightsquigarrow$ k (p_2) $\rightsquigarrow$ j, By Lemma 1(Lecture 12), p_1 is a shortest path from i to k with all intermediate vertices in the set {1, 2, . . . , k−1}. Similarly, p_2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2, . . . , k−1}.

The Floyd-Warshall Algorithm

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, . . . , k}. When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

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

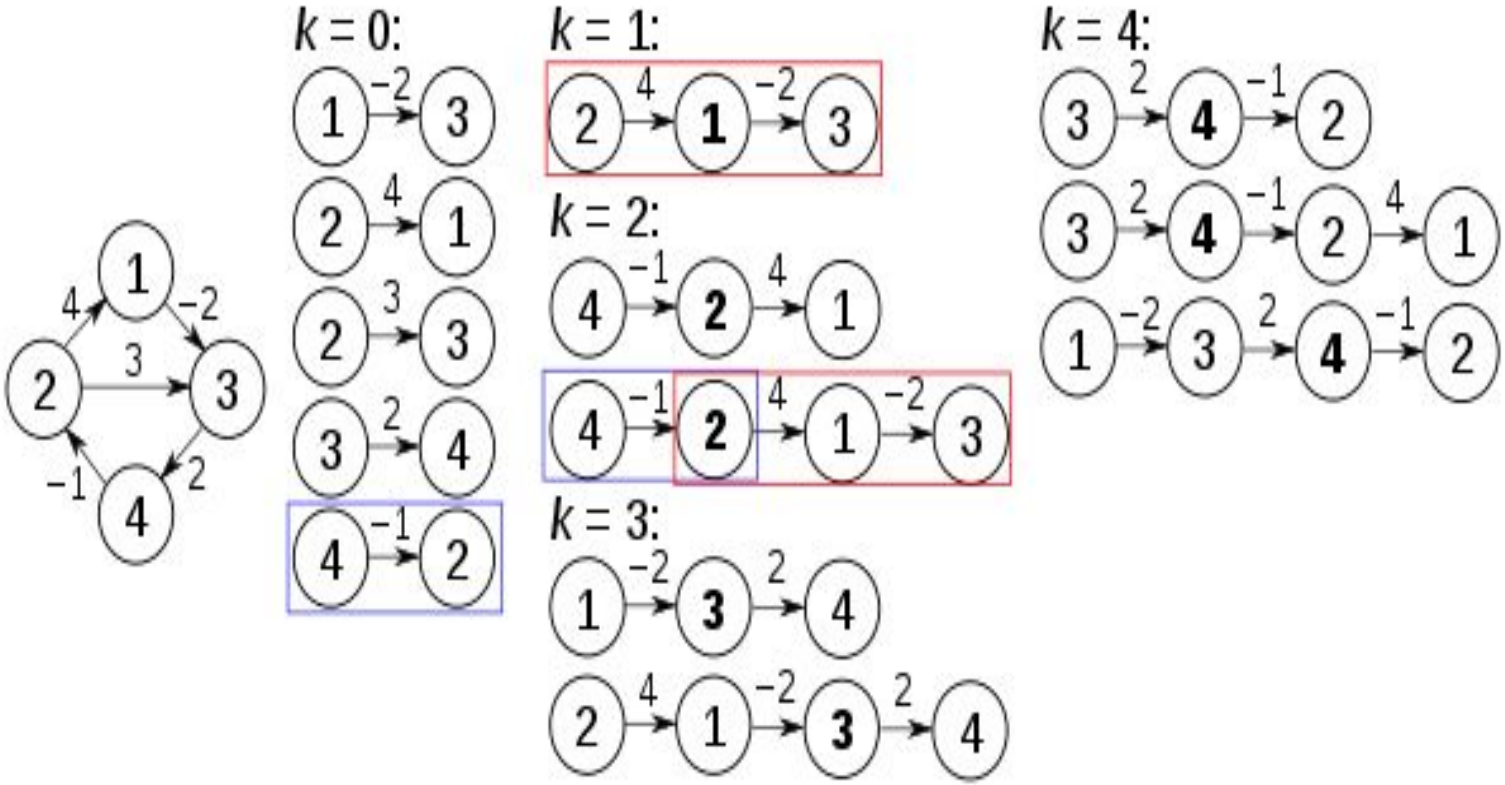
Because for any path, all intermediate vertices are in the set {1, 2, . . . , n}, the matrix D(n) = $d_{ij}^{(n)}$ gives the final answer:

$$d_{ij}^{(n)} = \delta(i, j) \text{ for all } i, j \in V$$

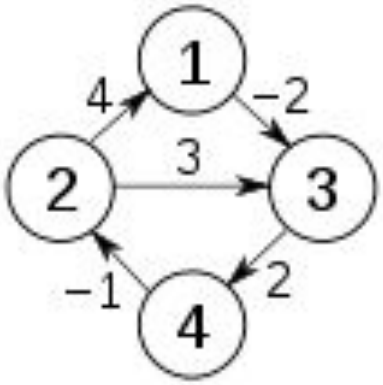
The Floyd-Warshall Algorithm

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

An Example



An Example (Matrices)



$k=0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=1$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=2$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k=3$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k=4$		j			
		1	**2**	**3**	**4**
i	**1**	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

The Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops.

Because each execution of line 7 takes O(1) time, the algorithm runs in time $\Theta(n^3)$.

As the previous dynamic program, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small.

Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

The Floyd-Warshall Algorithm

Now we consider how to construct a shortest path.

We need to define a predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either i = j or there is no path from i to j, otherwise π_{ij} is the predecessor of j on some shortest path from i. To obtain Π, we compute a sequence of matrices $\Pi(0), \Pi(1), \ldots, \Pi(n)$, where $\Pi = \Pi(n)$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from j on a shortest path from vertex i with all intermediate vertices in the set {1, 2, . . . , k}. Then we have

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

The Floyd-Warshall Algorithm

For k ≥ 1, if we take the path i ⇝ k ⇝ j, where k ≠ j, , then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set {1, 2, . . . , k − 1}. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set {1, 2, . . . , k − 1}.

Formally, for k ≥ 1,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

The Floyd-Warshall Algorithm

For each vertex $i \in V$, define the predecessor subgraph of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \text{ and } E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\}.$$

If $G_{\pi,i}$ is a shortest-paths tree, then we can use the following procedure to print a shortest path from vertex i to vertex j.

The Floyd-Warshall Algorithm

```
Print-All-Pairs-Path(Π, i, j).
1:  if i == j then
2:      print i
3:  else if π_ij == NIL then
4:      print "no path from i to j exists"
5:  else
6:      Print-All-Pairs-Shortest-Path(Π, i, π_ij )
7:      print j
```

For the Π from the Floyd-Warshall algorithm, it can be proved that $G_{\pi,i}$ is a shortest path tree with root i.

Transitive Closure

If we understand Floyd-Warshall algorithm well, the following algorithm for the transitive closure of a directed graph becomes really easy. The transitive closure of a directed graph G = (E, V), which is a graph $G^* = (V, E^*)$, where
$E^* = \{(i, j)$: there is a path from vertex i to vertex j in G}.
One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.
There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time that can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and + in the Floyd-Warshall algorithm.

Transitive Closure

For i, j, k = 1, 2, . . . , n, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set {1, 2, . . . , k}, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence on Slide 31, is

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i,j) \notin E, \\ 1 & \text{if } i = j \text{ or } (i,j) \in E. \end{cases}$$

and for k ≥ 1,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Transitive Closure

We compute the matrices T (k) = ($t_{ij}^{(k)}$) in order of increasing k

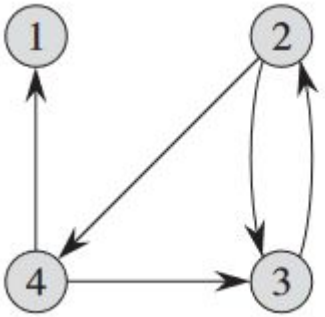
```
TRANSITIVE-CLOSURE(G)
 1  n = |G.V|
 2  let T^(0) = (t_ij^(0)) be a new n × n matrix
 3  for i = 1 to n
 4      for j = 1 to n
 5          if i == j or (i, j) ∈ G.E
 6              t_ij^(0) = 1
 7          else t_ij^(0) = 0
 8  for k = 1 to n
 9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
13  return T^(n)
```

Transitive Closure

The above procedure also runs in $\Theta(n^3)$ time. But on some computers, logical operations on single-bit values execute faster than arithmetic operations on integer words of data.

Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the Floyd-Warshall Algorithm's by a factor corresponding to the size of a word of computer storage.

Transitive Closure



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

History of Floyd-Warshall Algorithm

From Wiki:

The Floyd–Warshall algorithm is an example of dynamic programming, and was published in its currently recognized form by Robert Floyd in 1962.[3] However, it is essentially the same as algorithms previously published by Bernard Roy in 1959[4] and also by Stephen Warshall in 1962[5] for finding the transitive closure of a graph,[6] and is closely related to Kleene's algorithm (published in 1956) for converting a deterministic finite automaton into a regular expression.[7] The modern formulation of the algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.[8]

After Class

- After class reading: Part VI 25.2

Amortized Analysis (Chapter 17)

Introduction

In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.

With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.

An amortized analysis guarantees the average performance of each operation in the worst case.

Introduction (Cont.)

The key idea behind amortized analysis is to spread the cost of an expensive operation over several operations. For example, consider a dynamic array data structure that is resized when it runs out of space. The cost of resizing the array is expensive, but it can be amortized over several insertions into the array, so that the average time complexity of an insertion operation is constant.

Amortized analysis is useful for designing efficient algorithms for data structures such as dynamic arrays, priority queues, and disjoint-set data structures. It provides a guarantee that the average-case time complexity of an operation is constant, even if some operations may be expensive.

Topic I: Aggregate Analysis

In *aggregate analysis*, we show that for all n, a sequence of n operations takes worst-case time T(n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore T(n)/n.

Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.

Stack operations

We have already defined the stack operations: Push(S, x) and Pop(S). each of these operations runs in O.1/ time, let us consider the cost of each to be 1.

Topic I: Aggregate Analysis

Now we consider another operation:

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```

The stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects.

The actual running time is linear in the number of Pop operations actually executed, and thus we can analyze Multipop in terms of the abstract costs of 1 each for Push and Pop. The number of iterations of the while loop is the number min(s, k) of objects popped off the stack.

Topic I: Aggregate Analysis

Next we consider a procedure that consists of a sequence of n Push, Pop and Multipop operations on an initially empty stack.

The worst-case cost of a Multipop operation in the sequence is O(n), since the stack size is at most n. The worst-case time of any stack operation is therefore O(n), and hence a sequence of n operations costs $O(n^2)$, since we may have O(n) MULTIPOP operations costing O(n) each.

Although this analysis is correct, the $O(n^2)$ result, which we obtained by considering the worst-case cost of each operation individually, is not tight.

Topic I: Aggregate Analysis

Using *aggregate analysis*, we can obtain a better upper bound that considers the entire sequence of n operations. In fact, although a single Multipop operation can be expensive, any sequence of n Push,Pop and Multipop operations on an initially empty stack can cost at most O(n).

We can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the # of times that Pop can be called on a nonempty stack, including calls within Multipop, is at most the # of Push operations, which is at most n.

For any value of n, any sequence of n Push, Pop and Multipop operations takes a total of O(n) time. The average cost of an operation is O(n)/n = O(1). So in this example, all three stack operations have an amortized cost of O(1).

Topic I: Aggregate Analysis

Incrementing a binary counter:

Consider the problem of implementing a k-bit binary counter that counts upward from 0. We use an array A[0 . . . k − 1] of bits, where A.length = k, as the counter. Initially, x = 0, so A[i] = 0 for i = 0, . . . , k−1. To add 1 (modulo 2^k) to the value in the counter, we use the following procedure.

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

Topic I: Aggregate Analysis

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right.Notice that the total cost is always less than twice the total number of INCREMENT operations.

Topic I: Aggregate Analysis

In the worst case, the while loop in above procedure takes time Θ(k). Thus, a sequence of n Increment operations on an initially zero counter takes time O(nk) in the worst case.

A[0] does flip each time Increment is called. The next bit up, A[1], flips only every other time: a sequence of n Increment causes A[1] flip $\lfloor n/2 \rfloor$ times. Similarly, A[2] flips only every fourth time. In general, A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n Increment operations on an initially zero counter. The total number of flips is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

The worst-case time for a sequence of n Increment operations on an initially zero counter is therefore O(n). The average cost of each operation, and therefore the amortized cost per operation, is O(n)/n = O(1).

Topic I: Aggregate Method

In the worst case, the while loop in above procedure takes time Θ(k). Thus, a sequence of n Increment operations on an initially zero counter takes time O(nk) in the worst case.

A[0] does flip each time Increment is called. The next bit up, A[1], flips only every other time: a sequence of n Increment causes A[1] flip $\lfloor n/2 \rfloor$ times. Similarly, A[2] flips only every fourth time. In general, A[i] flips $\lfloor n/2^i \rfloor$ times in a sequence of n Increment operations on an initially zero counter. The total number of flips is

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

The worst-case time for a sequence of n Increment operations on an initially zero counter is therefore O(n). The average cost of each operation, and therefore the amortized cost per operation, is O(n)/n = O(1).

After Class

- After class reading: Part IV 17.1