COMP 761: Lecture 32 – Miscellaneous

David Rolnick

November 18, 2020

Problem

Find a situation where Ford-Fulkerson can take either 2 steps or 2 million steps depending on how you pick the augmenting paths.

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Additional context for Problem 3 (see later in class).

Course Announcements

- Additional context for Problem 3 (see later in class).

Course Announcements

- Additional context for Problem 3 (see later in class).
- Office hours: Vincent at **11 am** Thurs (different time this week), David at 10 am Fri.

Review: Dynamic programming

Review: Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.

Review: Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.
- One way of doing this is to solve all the possible subproblems in order of increasing size, so each one has all the solutions it needs.

Review: Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.
- One way of doing this is to solve all the possible subproblems in order of increasing size, so each one has all the solutions it needs.
- Alternatively, they can be computed as they are needed, but then stored in case they must be reused. This is called *memoization* (not a typo – we are making *memos* of useful solutions).

Review: Dynamic programming

- *Dynamic programming* is a paradigm of algorithm design in which solutions to smaller subproblems are calculated *and reused*.
- One way of doing this is to solve all the possible subproblems in order of increasing size, so each one has all the solutions it needs.
- Alternatively, they can be computed as they are needed, but then stored in case they must be reused. This is called *memoization* (not a typo – we are making *memos* of useful solutions).
- Why is this called "dynamic"? Richard Bellman chose this word because:

> *It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. ... Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.*

Matrix multiplication

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.
- So naïve time is mnp operations.

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.
- So naïve time is mnp operations.
- Now suppose we multiply matrices A, B, C of sizes 10×100, 100×5, and 5×50.

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.
- So naïve time is mnp operations.
- Now suppose we multiply matrices A, B, C of sizes 10×100, 100×5, and 5×50.
- How long does it take?

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.
- So naïve time is mnp operations.
- Now suppose we multiply matrices A, B, C of sizes 10×100, 100×5, and 5×50.
- How long does it take?
- If we multiply $(AB)C$, then we have $(10)(100)(5) = 5000$ for the 10×5 matrix AB and $(10)(5)(50) = 2500$ operations for the 10×50 matrix $(AB)C$, for a total of

$$5000 + 2500 = 7500.$$

Matrix multiplication

- How fast is a naïve algorithm to multiply matrices A of size $m \times n$ and B of size $n \times p$?
- The result is a matrix of size $m \times p$, with each entry involving a sum of n entries.
- So naïve time is mnp operations.
- Now suppose we multiply matrices A, B, C of sizes 10×100, 100×5, and 5×50.
- How long does it take?
- If we multiply $(AB)C$, then we have $(10)(100)(5) = 5000$ for the 10×5 matrix AB and $(10)(5)(50) = 2500$ operations for the 10×50 matrix $(AB)C$, for a total of

$$5000 + 2500 = 7500.$$

- If we multiply $A(BC)$, then we have $(100)(5)(50) = 25000$ for the 100×50 matrix BC and $(10)(100)(50) = 50000$ for the 10×50 matrix $A(BC)$, for a total of

$$25000 + 50000 = 75000.$$

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.
- Let's think about the last product we are doing – product of first k matrices, multiplied by product of remaining $n-k$ matrices, for some $k = 1, \ldots, (n-1)$:

$$P_n = \sum_{k=1}^{n-1} P_k P_{n-k}.$$

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.
- Let's think about the last product we are doing – product of first k matrices, multiplied by product of remaining $n-k$ matrices, for some $k = 1, \ldots, (n-1)$:

$$P_n = \sum_{k=1}^{n-1} P_k P_{n-k}.$$

- Roughly how big is P_n?

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.
- Let's think about the last product we are doing – product of first k matrices, multiplied by product of remaining $n-k$ matrices, for some $k = 1, \ldots, (n-1)$:

$$P_n = \sum_{k=1}^{n-1} P_k P_{n-k}.$$

- Turns out the solution is the Catalan numbers: $1, 1, 2, 5, 14, 42, 132, \ldots$

Problem

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- We could just try all possible ways to do it, e.g.

$$A_1(A_2(A_3A_4))$$
$$A_1((A_2A_3)A_4)$$
$$(A_1A_2)(A_3A_4)$$
$$(A_1(A_2A_3))A_4$$
$$((A_1A_2)A_3)A_4$$

- Let P_n be the # of ways to parenthesize the product of n matrices.
- Let's think about the last product we are doing – product of first k matrices, multiplied by product of remaining $n-k$ matrices, for some $k = 1, \ldots, (n-1)$:

$$P_n = \sum_{k=1}^{n-1} P_k P_{n-k}.$$

- Turns out the solution is the Catalan numbers: $1, 1, 2, 5, 14, 42, 132, \ldots$
- Grow exponentially, so trying all of these possibilities is a bad idea.

Sidenote: Catalan numbers

Sidenote: Catalan numbers

- Sequence of numbers $C_0, C_1, C_2, \ldots = 1, 1, 2, 5, 14, 42, 132, \ldots$

Sidenote: Catalan numbers

- Sequence of numbers $C_0, C_1, C_2, \ldots = 1, 1, 2, 5, 14, 42, 132, \ldots$
- Satisfy the recurrence relation:

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

Sidenote: Catalan numbers

- Sequence of numbers $C_0, C_1, C_2, \ldots = 1, 1, 2, 5, 14, 42, 132, \ldots$
- Satisfy the recurrence relation:

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

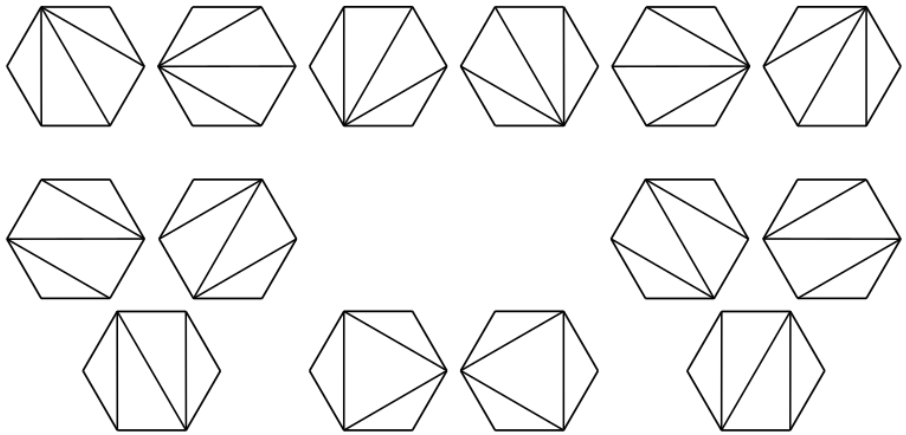
- (We were looking at $P_n = C_{n-1}$.)
- C_n also equals the number of binary trees with $n+1$ leaves.

Sidenote: Catalan numbers

- Sequence of numbers $C_0, C_1, C_2, \ldots = 1, 1, 2, 5, 14, 42, 132, \ldots$
- Satisfy the recurrence relation:

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

- (We were looking at $P_n = C_{n-1}$.)
- C_n also equals the number of binary trees with $n+1$ leaves.
- Or the number of ways to triangulate a convex polygon with $n+2$ sides:

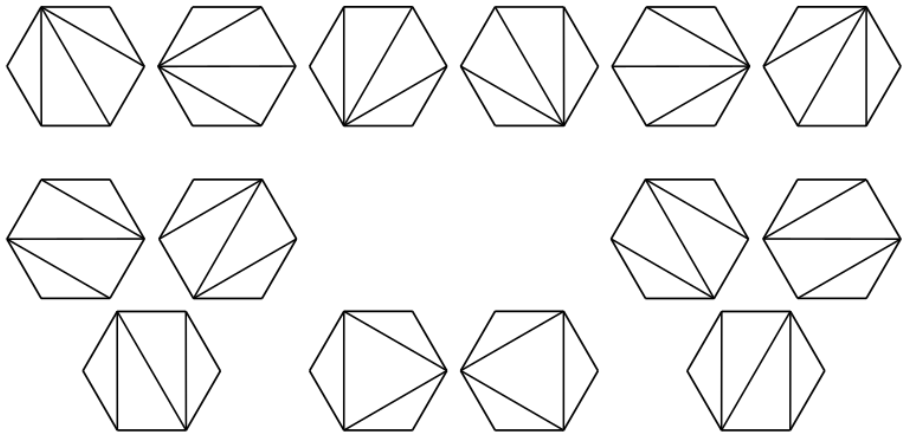


Sidenote: Catalan numbers

- Sequence of numbers $C_0, C_1, C_2, \ldots = 1, 1, 2, 5, 14, 42, 132, \ldots$
- Satisfy the recurrence relation:

$$C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}.$$

- (We were looking at $P_n = C_{n-1}$.)
- C_n also equals the number of binary trees with $n+1$ leaves.
- Or the number of ways to triangulate a convex polygon with $n+2$ sides:



- Closed form expression:

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.
- The parenthesization of the first k and last $n - k$ must both be optimal solutions to subproblems.

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.
- The parenthesization of the first k and last $n - k$ must both be optimal solutions to subproblems.
- For example, if we know the optimal way to multiply $A_1, A_2, \ldots, A_{10}$ is to first multiply A_1 through A_6 (in some order) and then multiply that by $A_7 A_8 A_9 A_{10}$ (computed in some order), then we can use the optimal parenthesizations for A_1 through A_6 and for A_7 through A_{10}.

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.
- The parenthesization of the first k and last $n - k$ must both be optimal solutions to subproblems.
- For example, if we know the optimal way to multiply $A_1, A_2, \ldots, A_{10}$ is to first multiply A_1 through A_6 (in some order) and then multiply that by $A_7 A_8 A_9 A_{10}$ (computed in some order), then we can use the optimal parenthesizations for A_1 through A_6 and for A_7 through A_{10}.
- Suppose the optimal parenthesization of $A_1 A_2 \cdots A_n$ takes $f(A_1, A_2, \ldots, A_n)$ operations.

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.
- The parenthesization of the first k and last $n - k$ must both be optimal solutions to subproblems.
- For example, if we know the optimal way to multiply $A_1, A_2, \ldots, A_{10}$ is to first multiply A_1 through A_6 (in some order) and then multiply that by $A_7 A_8 A_9 A_{10}$ (computed in some order), then we can use the optimal parenthesizations for A_1 through A_6 and for A_7 through A_{10}.
- Suppose the optimal parenthesization of $A_1 A_2 \cdots A_n$ takes $f(A_1, A_2, \ldots, A_n)$ operations.
- We have

$$f(A_1, \ldots, A_n) = \min_k \left(f(A_1, \ldots, A_k) + p_1 p_{k+1} p_{n+1} + f(A_{k+1}, \ldots, A_n) \right).$$

Dynamic programming approach

Suppose we are given matrices $A_1, A_2, \ldots, A_n$ of dimensions $p_1 \times p_2$, $p_2 \times p_3$, $\ldots$, $p_n \times p_{n+1}$. What is an efficient algorithm for working out the most efficient order in which to multiply the matrices?

- How can we use a dynamic programming approach?
- We know we can split a product into first k and last $n - k$ for some k.
- The parenthesization of the first k and last $n - k$ must both be optimal solutions to subproblems.
- For example, if we know the optimal way to multiply $A_1, A_2, \ldots, A_{10}$ is to first multiply A_1 through A_6 (in some order) and then multiply that by $A_7 A_8 A_9 A_{10}$ (computed in some order), then we can use the optimal parenthesizations for A_1 through A_6 and for A_7 through A_{10}.
- Suppose the optimal parenthesization of $A_1 A_2 \cdots A_n$ takes $f(A_1, A_2, \ldots, A_n)$ operations.
- We have

$$f(A_1, \ldots, A_n) = \min_k \left(f(A_1, \ldots, A_k) + p_1 p_{k+1} p_{n+1} + f(A_{k+1}, \ldots, A_n) \right).$$

- So we can precompute the solutions to these subproblems.

Dynamic programming approach

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- What is a good base case?

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

$$f(A_i, \ldots, A_j) = \min_{i \leq k \leq j-1} \left(f(A_i, \ldots, A_k) + p_i p_{k+1} p_{j+1} + f(A_{k+1}, \ldots, A_j) \right).$$

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

$$f(A_i, \ldots, A_j) = \min_{i \leq k \leq j-1} \left(f(A_i, \ldots, A_k) + p_i p_{k+1} p_{j+1} + f(A_{k+1}, \ldots, A_j) \right).$$

- We already know $f(A_i, \ldots, A_k)$ and $f(A_{k+1}, \ldots, A_j)$ by the inductive hypothesis, so this takes time $\Theta(j - i)$ (since $j - i$ terms in the min).

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

$$f(A_i, \ldots, A_j) = \min_{i \leq k \leq j-1} \left(f(A_i, \ldots, A_k) + p_i p_{k+1} p_{j+1} + f(A_{k+1}, \ldots, A_j) \right).$$

- We already know $f(A_i, \ldots, A_k)$ and $f(A_{k+1}, \ldots, A_j)$ by the inductive hypothesis, so this takes time $\Theta(j - i)$ (since $j - i$ terms in the min).
- This completes the induction.

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

$$f(A_i, \ldots, A_j) = \min_{i \leq k \leq j-1} \left(f(A_i, \ldots, A_k) + p_i p_{k+1} p_{j+1} + f(A_{k+1}, \ldots, A_j) \right).$$

- We already know $f(A_i, \ldots, A_k)$ and $f(A_{k+1}, \ldots, A_j)$ by the inductive hypothesis, so this takes time $\Theta(j - i)$ (since $j - i$ terms in the min).
- This completes the induction.
- What is the total time?

Dynamic programming approach

- We're only going to need products of *consecutive* matrices.
- Suppose for each i, j, we have precomputed $f(A_i, A_{i+1}, \ldots, A_j)$.
- Let's use induction on how many matrices: suppose we can do this for any i, j with $j - i \leq m$, now show we can do it for $j - i = m + 1$.
- Base case of $j - i = 0$ is easy since $f(A_i) = 0$ (just one matrix).
- Now suppose for $j - i \leq m$, prove for $j - i = m + 1$.

$$f(A_i, \ldots, A_j) = \min_{i \leq k \leq j-1} \left(f(A_i, \ldots, A_k) + p_i p_{k+1} p_{j+1} + f(A_{k+1}, \ldots, A_j) \right).$$

- We already know $f(A_i, \ldots, A_k)$ and $f(A_{k+1}, \ldots, A_j)$ by the inductive hypothesis, so this takes time $\Theta(j - i)$ (since $j - i$ terms in the min).
- This completes the induction.
- What is the total time?

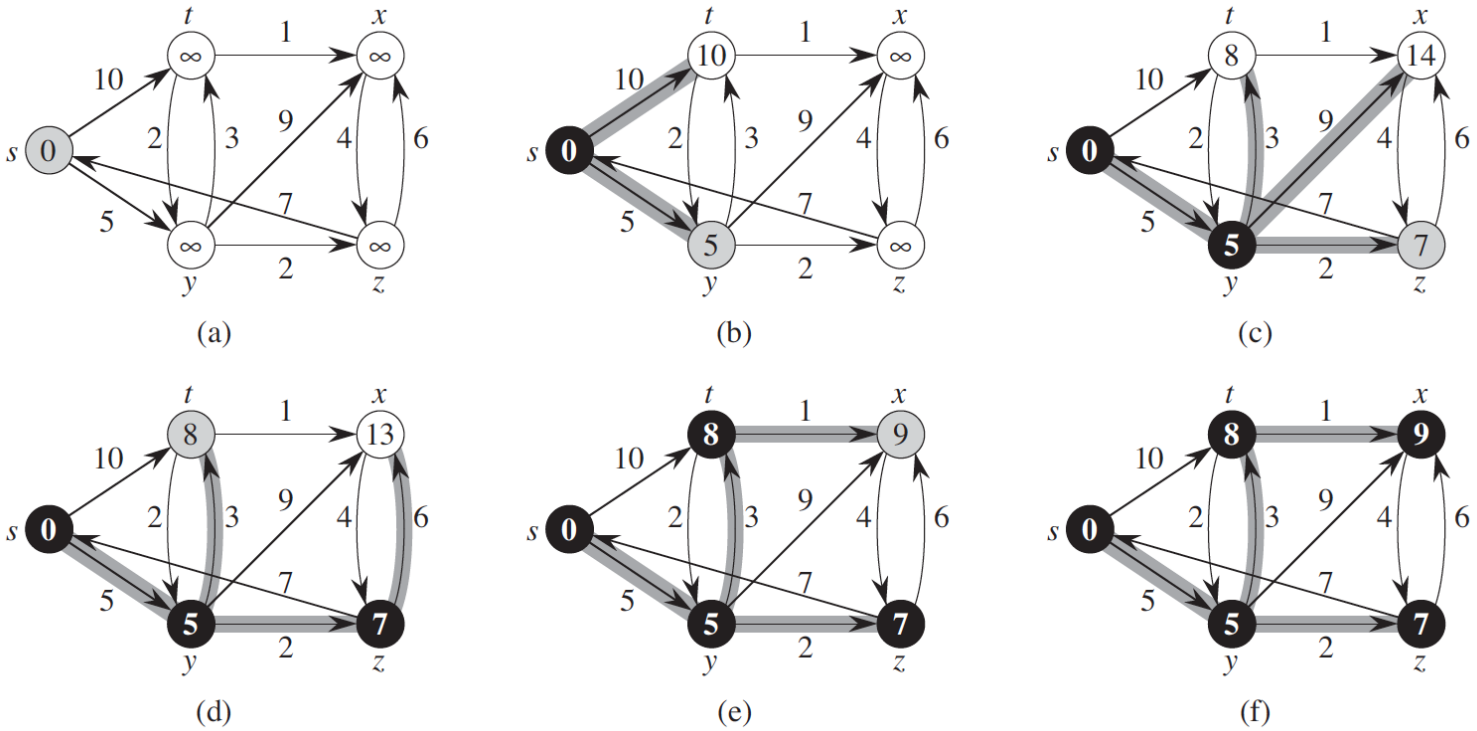
$$\sum_{i=1}^{n} \sum_{j=i}^{n} \Theta(j - i) = \sum_{i=1}^{n} \sum_{j=0}^{n-i} \Theta(j) = \sum_{i=1}^{n} \Theta((n - i)^2) = \Theta(n^3).$$

where we use $1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$ from the start of the course.

Shortest paths

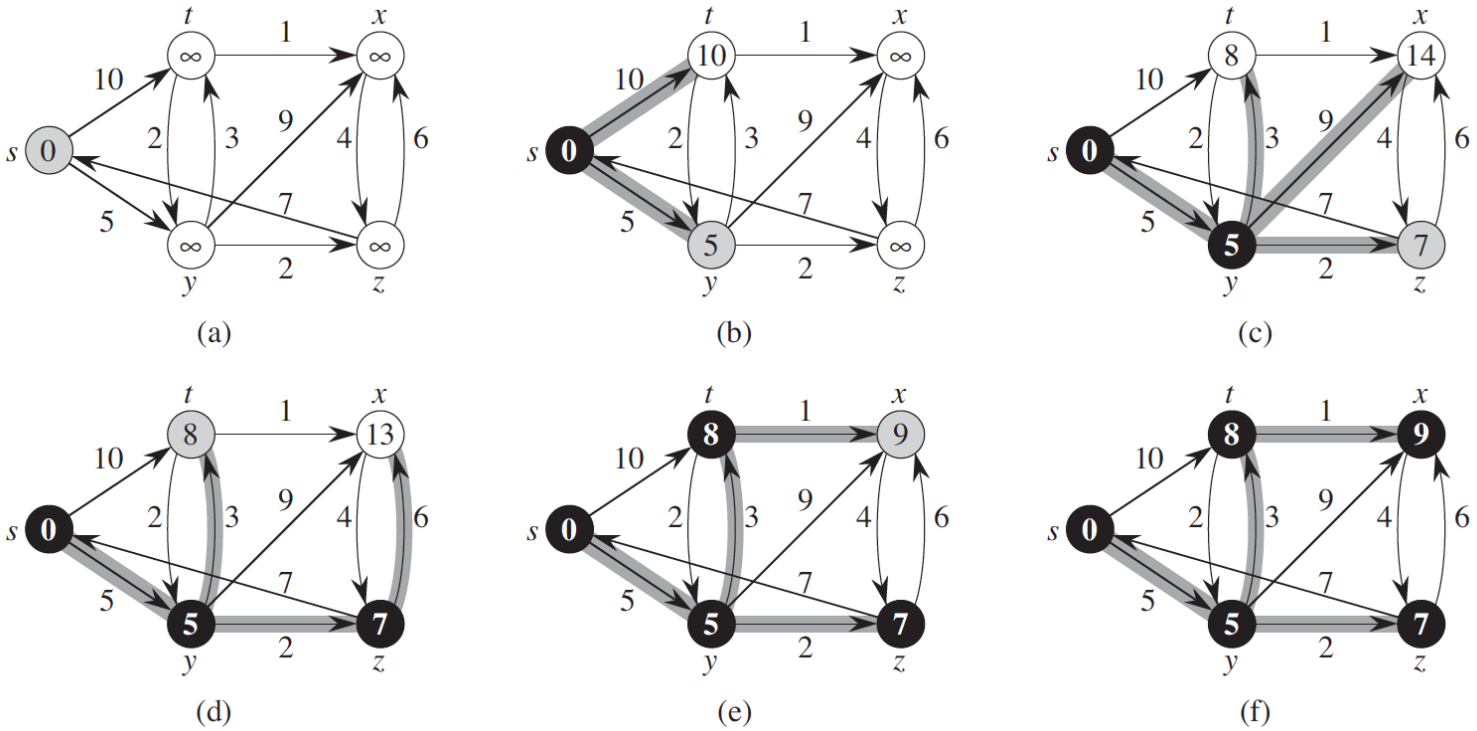
Shortest paths

- We were essentially using dynamic programming in Dijkstra's algorithm.



Shortest paths

- We were essentially using dynamic programming in Dijkstra's algorithm.



- Key property: If P is a shortest path between u and v and w is an intermediate vertex on that path, we can break P at w into path P_1 and P_2 where P_1 is a shortest path from u to w and P_2 is a shortest path from w to v.

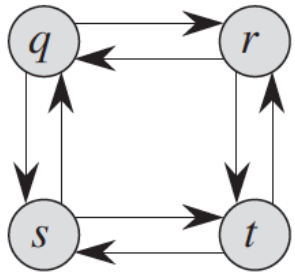
Longest paths

Longest paths

- Can we do a similar dynamic programming approach to find the *longest path* between u and v without any repeated vertices?

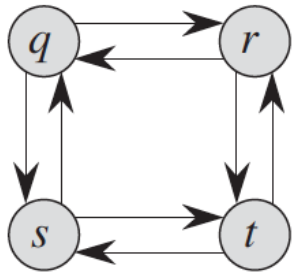
Longest paths

- Can we do a similar dynamic programming approach to find the *longest path* between u and v without any repeated vertices?
- No!



Longest paths

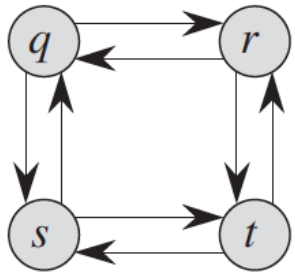
- Can we do a similar dynamic programming approach to find the *longest path* between u and v without any repeated vertices?
- No!



- In the graph above, (q, r, t) is a longest path from q to t, but if we break it into (q, r) and (r, t), neither of these is a longest path ((q, s, t, r) and (r, q, s, t) are longer, respectively).

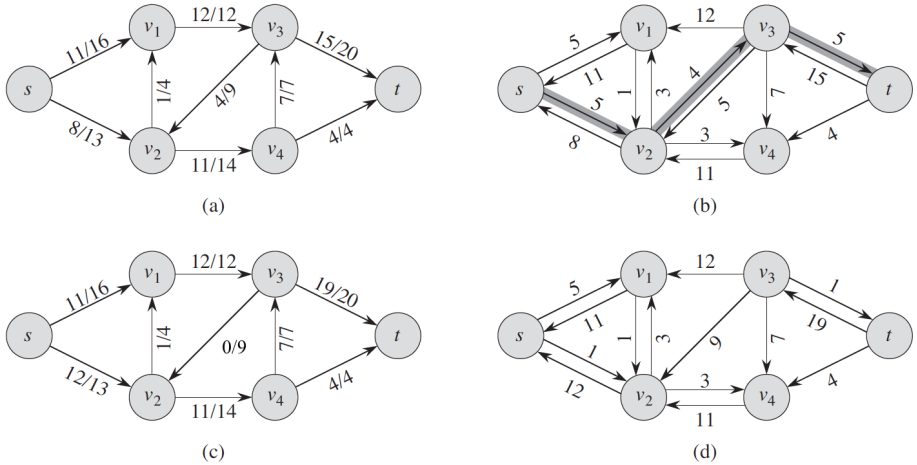
Longest paths

- Can we do a similar dynamic programming approach to find the *longest path* between u and v without any repeated vertices?
- No!

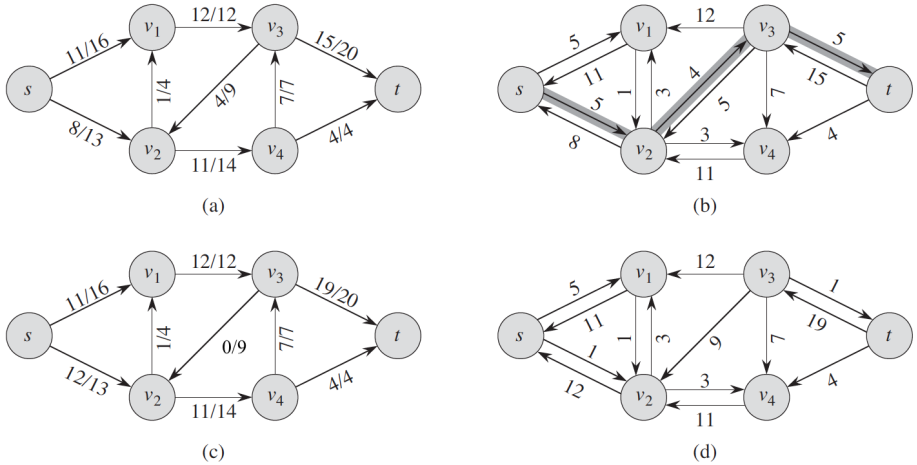


- In the graph above, (q, r, t) is a longest path from q to t, but if we break it into (q, r) and (r, t), neither of these is a longest path ((q, s, t, r) and (r, q, s, t) are longer, respectively).
- Dynamic programming only works if we can dissect a solution into the solutions of smaller problems.

Context for Problem 3 – augmenting paths in max flow

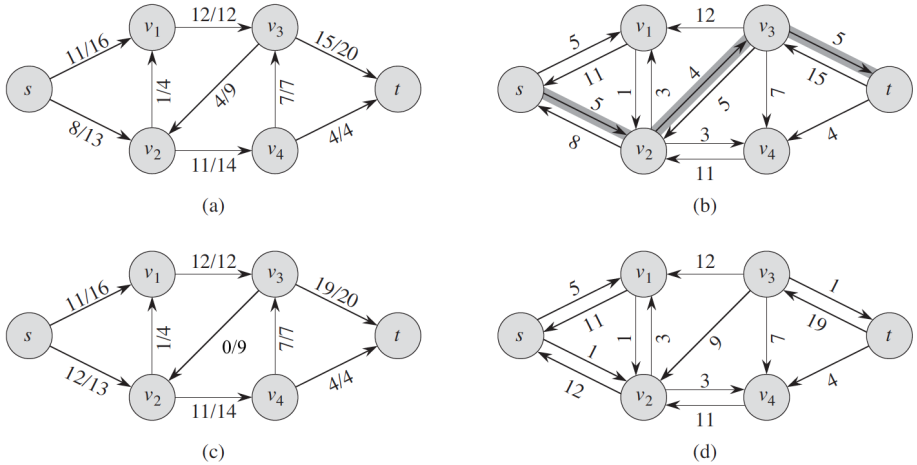


Context for Problem 3 – augmenting paths in max flow



- Given a flow network G and a flow f in it, we can define the *residual network* G_f as follows:

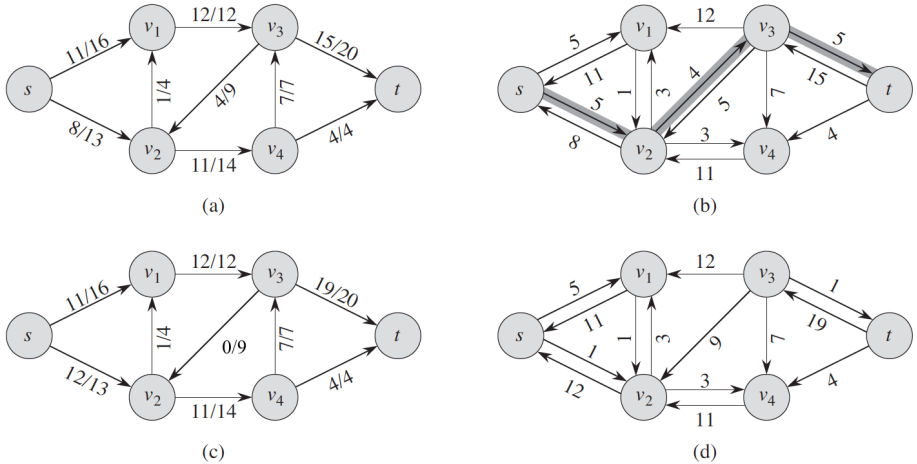
Context for Problem 3 – augmenting paths in max flow



- Given a flow network G and a flow f in it, we can define the *residual network* G_f as follows:
- If (u, v) is an edge in G, then both (u, v) and (v, u) are edges in G_f, with *residual capacities*

$$c_f(u, v) = c(u, v) - f(u, v)$$
$$c_f(v, u) = f(u, v).$$

Context for Problem 3 – augmenting paths in max flow

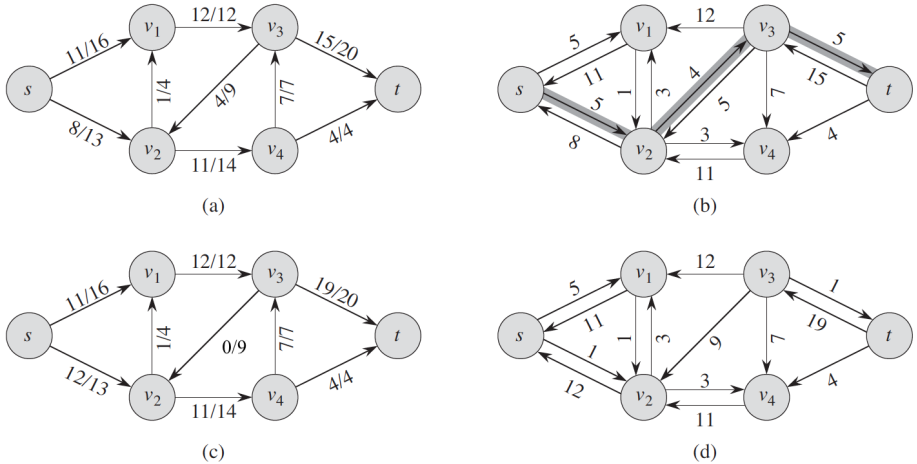


- Given a flow network G and a flow f in it, we can define the *residual network* G_f as follows:
- If (u, v) is an edge in G, then both (u, v) and (v, u) are edges in G_f, with *residual capacities*

$$c_f(u, v) = c(u, v) - f(u, v)$$
$$c_f(v, u) = f(u, v).$$

- Given a flow network G and a flow f from s to t, an *augmenting path* is a path from s to t in the residual network with pos. capacity along all edges.

Context for Problem 3 – augmenting paths in max flow



- Given a flow network G and a flow f in it, we can define the *residual network* G_f as follows:
- If (u, v) is an edge in G, then both (u, v) and (v, u) are edges in G_f, with *residual capacities*

$$c_f(u, v) = c(u, v) - f(u, v)$$
$$c_f(v, u) = f(u, v).$$

- Given a flow network G and a flow f from s to t, an *augmenting path* is a path from s to t in the residual network with pos. capacity along all edges.
- Each augmenting path gives a flow with the min capacity of all edges in the path. This flow can be added to the flow f to increase $|f|$.

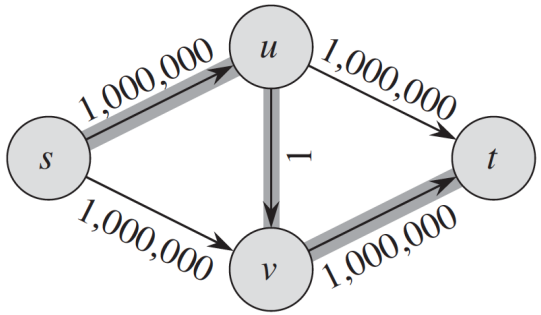
Context for Problem 3 – augmenting paths in max flow

Context for Problem 3 – augmenting paths in max flow

- Ford-Fulkerson works by iteratively finding an augmenting path in the residual network, adding it to the flow, recomputing the residual network, etc.

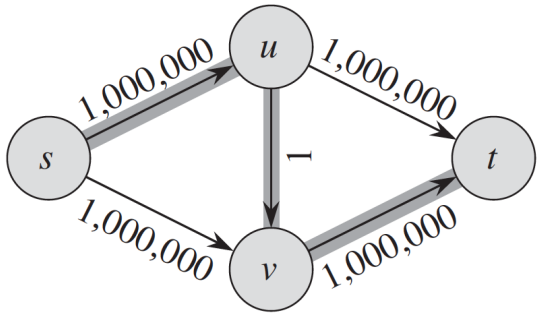
Context for Problem 3 – augmenting paths in max flow

- Ford-Fulkerson works by iteratively finding an augmenting path in the residual network, adding it to the flow, recomputing the residual network, etc.
- In this flow network, we could pick an augmenting path of capacity $1,000,000$ along the top, and another along the bottom. That would finish the algorithm in two steps.



Context for Problem 3 – augmenting paths in max flow

- Ford-Fulkerson works by iteratively finding an augmenting path in the residual network, adding it to the flow, recomputing the residual network, etc.
- In this flow network, we could pick an augmenting path of capacity $1,000,000$ along the top, and another along the bottom. That would finish the algorithm in two steps.



- We could also pick this bad sequence of augmenting paths, and take 2 million steps: