# Lecture Notes for Algorithms

Mingfu Shao
The Pennsylvania State University
November 6, 2024

# Contents

# Lecture 1 Introduction to Algorithms

## Algorithm, Problem, and Instance

What's an algorithm? Algorithm is a sequence of unambiguous specifications/instructions for solving a problem. Such specifications can be followed by a human, or implemented with a machine program. The *problem* studied in theoretical computer science needs to be formally defined, usually described using input and output.

For example, consider the *sorting problem*: its input is an array $A = [a_1, a_2, \cdots, a_n]$, and its output is the sorted array, say in ascending order, of $A$.

A problem describes a template. It's concrete, instantiated version is called an *instance*. For example, an instance for above sorting problem would be $A = [4, 2, 10, -5, 8]$, with the corresponding output being $[-5, 2, 4, 8, 10]$. The relationship between problem and instance is very similar to that between *class* and *object* in object-oriented programming languages. You may also think a problem is the set of all its possible instances.

Consider the so-called *selection problem*. Its input consists of an array $A = [a_1, a_2, \cdots, a_n]$ and an integer $k$, $1 \leq k \leq n$, and the output asks for the $k$-th smallest number of $A$. An instance of selection problem can be $A = [4, 2, 10, -5, 8]$ and $k = 2$, for which the output should be 2.

## Principles of Algorithm Design

There is always a straightforward algorithm to solve a problem, which is just enumerating all possible solution (and then pick the correct/best one). This is called *brute-force*. Brute-force may work for instances of size small, but in general it is *not efficient*.

We focus on designing *efficient* algorithms in this course. Here, "efficient" means "runs in polynomial-time", which we will formally define later in this course. There are two basic principles in designing efficient algorithms. Recall that an *algorithm* produces a *solution* for a *problem*.

1. *Partitioning problem into smaller subproblems.* This principle is from the perspective of problem. There are two strategies to reduce the size of a problem. First, we can partition a problem of size $n$ into smaller subproblems with size of $n-1$; this strategy leads to *dynamic programming* algorithms. Second, we can partition a problem of size $n$ into smaller subproblems with size of $n/2$; this strategy leads to *divide-and-conquer* algorithms.

2. *Iteratively Improving.* This principle is from the perspective of solution, i.e., we start with an initial/trivial solution, and we then gradually improve it to finally obtain the (optimal) solution. There are two strategies. First, we can start with a feasible but non-optimal solution, and then gradually improve it to achieve an optimal solutions; *linear programming* and *network flow* algorithms follow this strategy. Second, we can start with a partial-solution, and then gradually make it to a complete-solution; *greedy* algorithms follow this strategy.

## Relationship between Problems

We not only design algorithms for individual problems, but also study the relationship between problems. This is very important. Studying how problems relate help in the following three folds.

1. Allow us to build the hierarchy (i.e., classes of problems) and understand them. This is the targets of *theory of computational complexity*.

2. Allow us to solve new problems using existing algorithms. For example, if we know that problem $X$ can be transformed into problem $Y$ (we will formally define what does this mean) and we know a good algorithm for problem $Y$, and we know that we can use this algorithm for $Y$ to solve $X$.

3. Allow us to prove that a problem is hard to solve. For example, if we know that problem $X$ can be transformed into problem $Y$, and that there does not exist any efficient algorithhm for $X$, then we know that there does not exist efficient algorithm for problem $Y$.

## Example of Algorithm Design and Analysis

Let's consider the problem of *merging two sorted arrays*. The input of this problem is two sorted arrays, in ascending order, $A$ and $B$, and the output is a sorted array $C$ that consists of all elements in $A$ and $B$.

An instance of this problem is: $A = [-4, 2, 5, 8]$ and $B = [-3, 2, 3, 4]$. For this instance, the output should be $C = [-4, -3, 2, 2, 3, 4, 5, 8]$.

An algorithm usually requires *data structures* to store key intermediate information. In this case, we maintain two pointers, $k_A$ and $k_B$. Throughout the algorithm, we guarantee that $k_A$ and $k_B$ point to the smallest number in $A$ and $B$ that haven't been added to $C$.

The idea of the algorithm is to iteratively construct $C$ (so you may call it a greedy algorithm). In each step, we compare the two numbers at the two pointers: the smaller one of the two will be the smallest one in *all* numbers that haven't been added to $C$; we then add it to $C$ and update the pointers accordingly. The pseudo-code for this algorithm is given below (next page).

To analyze an algorithm, we need to (1) prove it's correct, and (2) analyze its running time.

We first prove this algorithm is correct, i.e., the resulting $C$ is sorted and includes all numbers in $A$ and $B$. This seems obvious. To give a formal and complete proof, we define the following *invariant*.

*Invariant:* for any $k = 0, 1, 2, \cdots, m+n$, at the end of the $k$-th iteration of above algorithm, we must have that $C$ stores the smallest $k$ numbers in $A$ and $B$, $k_A$ and $k_B$ points to the smallest number of $A \setminus C$ and $B \setminus C$, respectively.

We now prove above invariant is correct, by *induction*. The base case is $k = 0$. At this time (i.e., before the for-loop and after the initialization), $C$ is empty and $k_A$ and $k_B$ points to the first number in $A$ and $B$ respectively (and therefore the smallest one, as the given $A$ and $B$ are sorted). We now show the inductive step: assume that the invariant is correct for $k = 0, 1, \cdots, i$, we prove that the invariant is correct for $k = i+1$. The inductive assumption tells that at the end of the $i$-th iteration, $C[1 \cdots i]$ stores the first $i$ smallest numbers. Consider what the algorithm does in the $(i+1)$-th iteration: it compare the numbers at the pointers, and adds the smaller one to $C$. As each pointer now points to the smallest one in $A \setminus C$ and $B \setminus C$, we have that the smaller one is the $(i+1)$-th smallest number in $A$ and $B$. Besides, how the algorithm updates the pointers guarantees the pointers always point to the smallest one in $A \setminus C$ and $B \setminus C$, respectively. Therefore, the invariant holds after $(i+1)$-th iteration.

```
function merge-two-sorted-arrays (A[1···m], B[1···n])
    init an empty array C; (#units = 1)
    init pointers k_A = 1 and k_B = 1; (#units = 2)
    add a big number M (larger than any number in A and B) to the end of A and B: A[m+1] = M and
    B[n+1] = M; (think: why we do it?); (#units = 2)
    for k = 1 → m+n (advancing k from 1 to m+n; #units = m+n)
        if A[k_A] ≤ B[k_B] (#units = m+n)
            C[k] = A[k_A]; (#units = a)
            k_A = k_A + 1; (#units = a)
        else
            C[k] = B[k_B]; (#units = m+n−a)
            k_B = k_B + 1; (#units = m+n−a)
        end if;
    end for;
    return C; (#units = 1)
end algorithm;
```

We then analyze its running time. The running time is measured with the *basic computing units* used by the algorithm in the *worst case*. The definition of basic computing units depends on the computing model. Usually, we think each operation like assignment, basic arithmetic operation, comparison, etc, takes 1 unit. The worst case is w.r.t. all instances, i.e., the instance takes largest number of computing units.

The required units in each step of the algorithm is marked with blue text. Let $T(m,n)$ be the running time of this algorithm when $|A| = m$ and $|B| = n$. We can write $T(m,n) = 4m+4n+6$.

Note that the running time of an algorithm is a function of *input size*, rather than the *input*. The input size is usually the amount of memory needed to store the actually input. In this example, array of size $m$ can be stored in a block of memory with $m$ memory units. An implicit assumption here is that, every number in the given arrays can be stored in a single memory unit (for example, 32 bits to store an integer in C++). This is reasonable and widely used. In certain cases, where we study very large number (which cannot be stored in 32 bits for example), then we cannot assume that such number takes 1 memory unit; instead, a (large) number $x$ uses $\log_2 x$ bits in memory.

The running time of $T(m,n) = 4m+4n+6$ a brief, coarse estimation, as it depends on the computing model. Therefore, it doesn't make much sense to keep the coefficients 4 and the constants 5. Besides, we care how the running time grows as $m$ and $n$ grows. Hence, it suffices to write $T(m,n) = \Theta(m+n)$, meaning it grows as fast as function $m+n$. We will introduce asymptotic notations to facilitate this way of analyzing running time in next lecture.

# Lecture 2 Asymptotic Notations

## Definitions and Properties

**Definition 1** (Big-O)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers $n$. We say $f = O(g)$, if there exists positive number $c > 0$ and integer $N \geq 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq N$.

Similarly, we can define Big-O for multiple-variable functions.

**Definition 2** (Big-O)**.** Let $f = f(m,n)$ and $g = g(m,n)$ be two positive functions over integers $m$ and $n$. We say $f = O(g)$, if there exists positive number $c > 0$ and integers $M \geq 0$ and $N \geq 0$ such that $f(m,n) \leq c \cdot g(m,n)$ for all $m \geq M$ and $n \geq N$.

Intuitively, Big-O is analogous to "$\leq$". $f = O(g)$ means "$f$ grows no faster than $g$".

*Example.* Let $f(m,n) = 4m + 4n + 5$ and $g(m,n) = m + n$. We now show that $f = O(g)$, using above definition. To show it, we need to find $c$, $M$, and $N$. What are good choices for them? There are lots of choices; one set of it is: $c = 7$, $M = 1$, and $N = 1$. Let's verify: $f(m,n) - c \cdot g(m,n) = 4m + 4n + 5 - 7m - 7n = 5 - 3m - 3n \leq 5 - 3 - 3 = -1 \leq 0$, where we use that $m \geq M = 1$ and $n \geq N = 1$. This proves that $f = O(g)$.

**Definition 3** (Big-Omega)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers $n$. We say $f = \Omega(g)$, if there exists positive number $c > 0$ and integer $N \geq 0$ such that $f(n) \geq c \cdot g(n)$ for all $n \geq N$.

Similarly, we can define Big-Omega for multiple-variable functions.

**Definition 4** (Big-O)**.** Let $f = f(m,n)$ and $g = g(m,n)$ be two positive functions over integers $m$ and $n$. We say $f = \Omega(g)$, if there exists positive number $c > 0$ and integers $M \geq 0$ and $N \geq 0$ such that $f(m,n) \geq c \cdot g(m,n)$ for all $m \geq M$ and $n \geq N$.

Intuitively, Big-Omega is analogous to "$\geq$". $f = \Omega(g)$ means "$f$ grows at least as fast as $g$".

*Example.* Let $f(m,n) = 4m + 4n + 5$ and $g(m,n) = m + n$. We now show that $f = \Omega(g)$, using above definition. To show it, we need to find $c$, $M$, and $N$. We can choose: $c = 1$, $M = 0$, and $N = 0$. Let's verify: $f(m,n) - c \cdot g(m,n) = 4m + 4n + 5 - m - n = 5 + 3m + 3n \geq 5 \geq 0$, where we use that $m \geq M = 0$ and $n \geq N = 0$. This proves that $f = \Omega(g)$.

**Claim 1.** $f = O(g)$ if and only if $g = \Omega(f)$.

*Proof.* We have

$$
\begin{aligned}
& f = O(g) \\
\Leftrightarrow \quad & \exists\, c > 0, N \geq 0, \text{ s.t. } f(n) \leq c \cdot g(n), \forall n \geq N \\
\Leftrightarrow \quad & \exists\, c > 0, N \geq 0, \text{ s.t. } 1/c \cdot f(n) \leq g(n), \forall n \geq N \\
\Leftrightarrow \quad & \exists\, c' = 1/c > 0, N \geq 0, \text{ s.t. } g(n) \geq c' \cdot f(n), \forall n \geq N \\
\Leftrightarrow \quad & g = \Omega(f)
\end{aligned}
$$

□

**Definition 5** (Big-Theta)**.** We say $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

Intuitively, Big-Theta is analogous to "=". $f = \Theta(g)$ means "$f$ grows at the same rate as $g$".

*Example.* Let $f(m,n) = 4m + 4n + 5$ and $g(m,n) = m + n$. We have $f = \Theta(g)$ as we proved that both $f = O(g)$ and $f = \Omega(g)$.

**Definition 6** (small-o)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers $n$. We say $f =$

$o(g)$, if for every $c > 0$ there exists integer $N_c \geq 0$, where $N_c$ can be dependent on $c$, such that $f(n) \leq c \cdot g(n)$ for all $n \geq N_c$.

*Example.* Let $f(n) = n$ and $g(n) = n^2$. We now show that $f = o(g)$, using above definition. To prove it, we need to show that for any possible $c > 0$ there exists $N_c \geq 0$ such that $f(n) - c \cdot g(n) \leq 0$ when $n \geq N_c$. We write $f(n) - c \cdot g(n) = n - cn^2 = n(1 - cn)$. To let it be $\leq 0$, since $n \geq 0$, we can require $1 - cn \leq 0$, leading to $n \geq 1/c$. Therefore, we can choose $N_c = \lceil 1/c \rceil$. This completes the proof.

Intuitively, small-o is analogous to "$<$". $f = o(g)$ means $f$ grows (strictly) slower than $g$.

**Definition 7** (small-omega)**.** Let $f = f(n)$ and $g = g(n)$ be two positive functions over integers $n$. We say $f = \omega(g)$, if for every $c > 0$ there exists integer $N_c \geq 0$, where $N_c$ can be dependent on $c$, such that $f(n) \geq c \cdot g(n)$ for all $n \geq N_c$.

Intuitively, small-omega is analogous to "$>$". $f = \omega(g)$ means $f$ grows (strictly) faster than $g$.

Obviously, if $f = o(g)$ then $f = O(g)$; if $f = \omega(g)$ then $f = \Omega(g)$. This is intuitive, as "$<$" implies "$\leq$" and "$>$" implies "$\geq$". To formally see this, compare the definitions of small-o and big-O. $f = o(g)$ requires that $f(n) \leq c \cdot g(n)$, when $n \geq N_c$, for *every* possible $c > 0$. Therefore of course there exists *one* $c$ and $N_c$ such that $f(n) \leq c \cdot g(n)$, when $n \geq N_c$; this is all we need to prove $f = O(g)$. The same argument can be used for small-omega and big-Omega.

You might found that these asymptotic notations are similar to the (epsilon-delta)-defintions of limit. In fact, they are indeed closely related. Specifically, the limit of $f(n)/g(n)$, if exists (i.e., $f(n)/g(n)$ coverges as $n \to \infty$), or goes to infinity (i.e., $\lim_{n\to\infty} f(n)/g(n) = \infty$), we can conclude a relationship between $f$ and $g$:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \Rightarrow \quad f = o(g) \\ c > 0 & \Rightarrow \quad f = \Theta(g) \\ \infty & \Rightarrow \quad f = \omega(g) \\ \text{oscillate} & \Rightarrow \quad \text{no conclusion} \end{cases}$$

Above claim gives an convinent way to build asymptotic relationship. For the same example where $f(n) = n$ and $g(n) = n^2$. We now can show $f = o(g)$ by calculating $\lim_{n\to\infty} f(n)/g(n)$. In fact, $\lim_{n\to\infty} n/n^2 = \lim_{n\to\infty} 1/n = 0$. Hence, $f = o(g)$.

Another example: $f(n) = n^2$ and $g(n) = 2^n$. We calculate $\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} n^2/2^n$. Using L-Hopital rule, we have $\lim_{n\to\infty} n^2/2^n = \lim_{n\to\infty} 2n/(2^n \cdot \ln 2) = 2/(2^n \cdot \ln 2 \cdot \ln 2) = 0$. Hence, $f = o(g)$.

Note that when $f(n)/g(n)$ oscillates, as $n \to \infty$, then we cannot conclude anything. Note also that this reasoning is one-side. For example, if $f = \Theta(g)$ then we cannot guarantee that $\lim_{n\to\infty} f(n)/g(n) = c > 0$; for most functions this is correct but exceptions exist.

## Commonly-Used Functions in Algorithm Analysis

In theoretical computer science, we often see following categories of functions.

1. logarithmic functions: $\log\log n$, $\log n$, $(\log_n)^2$;
2. polynomial functions: $\sqrt{n} = n^{0.5}$, $n$, $n\log n$, $n^{1.001}$;
3. exponential functions: $2^n$, $n2^n$, $3^n$;
4. factorial functions: $n!$;

In above lists, any logarithmic function is small-o of any polynomial function: for example, $(\log n)^2 = o(n^{0.01})$; any polynomial function is small-o of any exponential function: for example, $n^2 = o(2^n)$; any exponential function is small-o of any factorial function: for example, $n2^n = o(n!)$. Within each category, a function to the left is small-o of a function to the right, for example $n \log n = o(n^{1.001})$.

# Lecture 3 Merge-Sort and Master's Theorem

## Framework of Divide-and-Conquer

We now start introducing the first algorithm-design technique: divide-and-conquer. A typical divide-and-conquer algorithm follows the framework below.

1. partition the original problem into smaller problems;
2. recursively solve all subproblems;
3. combine the solutions of the subproblems to obtain the solution of the original problem.

## Merge-Sort

We use sorting as the first problem to demonstrate designing divide-and-conquer algorithms. Recall that the *sorting* problem is to find the sorted array (say, in increasing order) $S'$ of a given array $S$. We now design a divide-and-conquer algorithm for it. For any recursive algorithm, we always need to clearly define the recursion. In this case, we define function merge-sort ($S$) returns the sorted array (in ascending order) of $S$.

The idea is to sort the first half and second half of $S$, by recursively calling the merge-sort function. How to obtain the sorted array of $S$ then with the two sorted half-sized arrays? We have introduced such an algorithm to merge two sorted arrays into a single sorted array. That's exactly the algorithm we need here in the combining step. The pseudo-code for the merge-sort function is given below.

```
Algorithm merge-sort (S[1 ··· n])
    if n ≤ 1: return S;
    S'_1 = merge-sort (S[1 ··· n/2]);
    S'_2 = merge-sort (S[n/2 + 1 ··· n]);
    return merge-two-sorted-arrays (S'_1, S'_2);
end algorithm;
```

## Analysis of Merge-Sort

The correctness of the algorithm can be proved by induction, as the natural structure of above algorithm is recursive. Specifically, we want to prove the statement that the merge-sort function with $A$ as input returns the sorted array of $A$ if $|A| = n$. The induction is w.r.t. $n$. The base case, i.e., $n = 1$, is clearly correct. In the inductive step, we assume that above statement is true for $|A| = 1, 2, \cdots, n-1$, and we aim to prove it is correct for $|A| = n$. As the algorithm is correct for $|A| = 1, 2, \cdots, n-1$, in particular, it is correct for $|A| = n/2$, i.e., $S'_1$ and $S'_2$ store the sorted array of $S_1$ and $S_2$, respectively. Combinining the correctness of merge-two-sorted-arrays that we have already proved, we have that merge-sort returns the sorted array of $S$.

To see its runing time, we define $T(n)$ as the running time of merge-sort ($S$) when $|S| = n$. Clearly, both merge-sort ($S[1 \cdots n/2]$) and merge-sort ($S[n/2+1 \cdots n]$) take $T(n/2)$ time. As merge-two-sorted-arrays takes $\Theta(|S'_1| + |S'_2|) = \Theta(n)$ time, we have the recurrence $T(n) = 2T(n/2) + \Theta(n)$. We will use *master's theorem* to get the closed form for $T(n)$, described below.

## Master's Theorem

Master's theorem gives closed form for the following recurrence:

$$T(n) = \begin{cases} aT(n/b) + \Theta(n^d) & \text{if } n \geq 2 \\ 1 & \text{if } n \leq 1 \end{cases}$$

Master's theorem is widely used to analyze the running time of divide-and-conquer algorithms. Recall that a divide-and-conquer algorithms first partition the original problems into subproblems, solve all subproblems recursively, and then combine them to answer the original question. Hence, above recurrence precisely describes the running time of such algorithms. Specifically, $a$ refers to the number of subproblems that the original problem is partitioned, $n/b$ refer to the input-size of each subproblem, and $\Theta(n^d)$ refer to the running time of the combining step. In merge-sort, $a = 2$, as it calls merge-sort twice in the algorithm, $b = 2$, as the input-size of each subproblem becomes $n/2$, and $d = 1$, as the merge-two-sorted-arrays takes $\Theta(n)$ time. Note that, it is not always the case that $a = b$ (in merge-sort though, $a = b$). Such example includes matrix multiplication, where $a = 4$ and $b = 2$.

We now solve above recurrence. Without loss of generality, we assume that $n$ is a power of $b$, i.e., $n = b^k$ for some $k$. To further simplify, we use $n^d$ instead of $\Theta(n^d)$. We therefore need to add $\Theta(\cdot)$ to the resulting formular of $T(n)$.

$$\begin{aligned} T(n) &= aT(n/b) + n^d \\ &= a(aT(n/b^2) + (n/b)^d) + n^d \\ &= a^2T(n/b^2) + a(n/b)^d + n^d \\ &= a^2(aT(n/b^3) + (n/b^2)^d) + a(n/b)^d + n^d \\ &= a^3T(n/b^3) + a^2(n/b^2)^d + a(n/b)^d + n^d \\ &= \cdots \\ &= a^kT(n/b^k) + \sum_{i=0}^{k-1} a^i(n/b^i)^d \end{aligned}$$

As we assume that $n = b^k$ and $T(1) = 1$, we have

$$T(n) = a^k + \sum_{i=0}^{k-1} a^i(n/b^i)^d = \sum_{i=0}^{k} a^i(n/b^i)^d = n^d \sum_{i=0}^{k} (a/b^d)^i.$$

Consider the following 3 cases.

1. If $a = b^d$, i.e., $d = \log_b a$, then $T(n) = n^d k = n^d \log_b n = \Theta(n \log n)$.
2. If $a < b^d$, i.e., $d > \log_b a$, then the series decreases exponentially, and therefore the item of $i = 0$ dominates. $T(n) = n^d a/b^d = \Theta(n^d)$.
3. If $a > b^d$, i.e., $d < \log_b a$, then the series increases exponentially, and therefore the item of $i = k$ dominates. $T(n) = n^d(a/b^d)^k = n^d a^k/b^{dk} = n^d a^k/n^d = a^k = a^{\log_b n} = n^{\log_b a}$.

We have used one facts about logarithmic function above: $a^{\log_b n} = n^{\log_b a}$.

Master's theorem can be summarized as below. For recurrence $T(n) = aT(n/b) + n^d$ with $T(1) = 1$, we have the following:

$$T(n) = \begin{cases} \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

In the case of merge-sort, $a = b = 2$ and $d = 1$. So $d = \log_b a = 1$. Hence, $T(n) = \Theta(n \log n)$.

A more generalized form of master's theorem is to solve this recurrence: $T(n) = aT(n/b) + n^d \log^s n$ with $T(1) = 1$. The closed form is given below:

$$T(n) = \begin{cases} \Theta(n^d \log^{s+1} n) & \text{if } d = \log_b a \\ \Theta(n^d \log^s n) & \text{if } d > \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

# Lecture 4 Counting Inversions

Let $A$ be an array of $n$ distinct integers. We say $(i, j)$ is an *inversion* in $A$ if $i < j$ and $A[i] > A[j]$. Given an array $A$, the problem of *counting inversions* seeks the number of inversions in $A$, formally written as $|\{(i, j) \mid i < j, A[i] > A[j]\}|$. For example, the number of inversions in array $A[4,3,7,2,5]$ is 5.

Clearly, a sorted array (of size $n$) in ascending order contains 0 inversion. A sorted array (of size $n$) in descending order contains $n(n-1)/2$ inversion. To see this, note that every pair $(i, j)$, $i < j$, forms an inversion because $A[i] > A[j]$; therefore, the number of inversions is: $\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n}(n-i) = n(n-1)/2$. Hence, the number of inversions of an array $A$ serves as a good quantiative measure that quantifies how far $A$ is from being (increasingly) sorted. Applications of this problem can be found in Section 5.3 [KT].

We now design algorithms to count inversions. A brute-force algorithm can be easily designed, that simply enumerates all pairs and counts the inversions:

```
Algorithm #1: brute-force-count-inv (A[1 ··· n])
    let inv = 0;
    for i = 1 to n
        for j = i + 1 to n
            if(A[i] > A[j]): inv = inv + 1;
        end for;
    end for;
    return inv;
end algorithm;
```

Clearly, above brute-force-count-inv algorithm takes $\Theta(n^2)$ time. Can we design a faster algorithm? Let's try the divide-and-conquer idea. We define a recursive function DC-count-inv ($A$) that returns the number of inversions in $A$. Naturally, we can divide array $A$ by splitting it in the middle, i.e., $A_1 = A[1 \cdots n/2]$ and $A_2 = A[n/2+1 \cdots n]$. The total number of inversions of $A$ consists of inversions in $A_1$, inversions in $A_2$, and inversions between $A_1$ and $A_2$. The first two parts can be calculated by recursively calling DC-count-inv($A_1$) and DC-count-inv($A_2$). To account for the third part, we introduce a new function count-inv-two-arrays ($S[1 \cdots m]$, $T[1 \cdots n]$), that calculates the inversions between $S$ and $T$, formally defined as $|\{(i, j) \mid 1 \le i \le m, 1 \le j \le n, S[i] > T[j]\}|$. We first give the pseudo-code for function DC-count-inv:

```
Algorithm #2: DC-count-inv (A[1 ··· n])
    if n ≤ 1: return 0;
    I1 = DC-count-inv (A[1 ··· n/2]);
    I2 = DC-count-inv (A[n/2 + 1 ··· n]);
    I3 = count-inv-two-arrays (A[1 ··· n/2], A[n/2 + 1 ··· n]);
    return I1 + I2 + I3;
end algorithm;
```

To complete above algorithm #2 we need to implement count-inv-two-arrays ($S$, $T$). Let's again use the "brute-force" idea, that compares all pairs between $S$ and $T$:

```
function count-inv-two-arrays (S[1···m], T[1···n])
    let inv = 0;
    for i = 1 to m
        for j = 1 to n
            if(S[i] > T[j]): inv = inv + 1;
        end for;
    end for;
    return inv;
end algorithm;
```

Function count-inv-two-arrays runs in $\Theta(mn)$ time. Hence, count-inv-two-arrays $(A[1\cdots n/2], A[n/2+1\cdots n])$ takes $\Theta(n/2 \cdot n/2) = \Theta(n^2)$ time. The running time of entire Algortithm #2, defined as $T(n)$, can be written as $T(n) = 2T(n) + \Theta(n^2)$. By using master's theorem, where $a = 2, b = 2, d = 2$, we conclude that $T(n) = \Theta(n^2)$.

Algorithm #2 is no better/faster than algorithm #1. They are essentially the same algorithm. Although algorithm #2 uses the divide-and-conquer framework, it in fact also enumerates all pairs. To see this, note that all pairs between $A[1\cdots n/2]$ and $A[n/2+1\cdots n]$ are compared by count-inv-two-arrays; all pairs within $A[1\cdots n/2]$ and all pairs within $A[n/2+1\cdots n]$ are also compared, as they recursively call count-inv-two-arrays which calls count-inv-two-arrays (on subarrays of $n/4$) as well. So, algorithm #2 enumerates all pairs. Hence, not a surprise, it's running time is $\Theta(n^2)$.

Can we really design faster algorithm for this problem? Let's borrow some ideas from merge-sort. Recall that, in merge-sort, the combining/merging step (i.e., function merge-two-sorted-arrays) takes $\Theta(n)$ time. And the reason is that, the two subarrays are already sorted, which is taken advantage of by the merging step. We therefore hope that, the inversions between two *sorted* arrays, can be counted in a faster way. We will demonstrate that, this is indeed true.

Here is algorithm #3. It counts the inversions *while* sorting the array. We define a recursive function DC-sort-count-inv $(A)$ that returns the sorted array of $A$ *and* the number of inversions in $A$. The framework of algorithm #3 is the same with that of algorithm #2. It's pseudo-code is given below, where the merging step, merge-count-inv-two-sorted-arrays, takes two *sorted* arrays as input, and return the merged/sorted array *and* the number of inversions between the two input arrays.

```
Algorithm #3: DC-sort-count-inv (A[1···n])
    if n ≤ 1: return (A, 0);
    (A'_1, I_1) = DC-sort-count-inv (A[1···n/2]);
    (A'_2, I_2) = DC-sort-count-inv (A[n/2+1···n]);
    (A', I_3) = merge-count-inv-two-sorted-arrays (A'_1, A'_2);
    return (A', I_1 + I_2 + I_3);
end algorithm;
```

The remaining (key) step is to implement function merge-count-inv-two-sorted-arrays $(S[1\cdots,m], T[1\cdots n])$. Again, we aim for doing it faster (than brute-force which takes $\Theta(mn)$ time) by making use of the fact that both $S$ and $T$ are already sorted. It's pseudo-code is given below; it just adds a single line to the merge-two-sorted-arrays function.

```
function merge-count-inv-two-sorted-arrays (S[1···m], T[1···n])
    init an emity array C;
    init pointers i = 1 and j = 1;
    init inv = 0;
    S[m+1] = M and T[n+1] = M;
    for k = 1 → m+n
        if S[i] ≤ T[j]
            C[k] = S[i];
            i = i+1;
        else
            C[k] = T[j];
            j = j+1;
            inv = inv + (m−i+1);
        end if;
    end for;
    return (C, inv);
end algorithm;
```

Let's explain/prove why above algorithm works. We have proved that $C$ is the merged, sorted array of $S$ and $T$ (in Lecture 01) so now we focus on showing that inv stores that correct number of inversions between $S$ and $T$, i.e., $|\{(i,j) \mid S[i] > T[j], 1 \le i \le m, 1 \le j \le n\}|$. The above algorithm maintains two pointers $i$ and $j$ that (always) point to the first number (also the smallest number) in $S$ and $T$ respectively that haven't been added to $C$. In case of $S[i] \le T[j]$, which means $(i,j)$ does not form an inversion, we do not increase *inv*. In case of $S[i] > T[j]$, which means $(i,j)$ forms an inversion, we increase *inv* by $m-i+1$. Why? This is because $S$ is sorted, i.e., $S[i] < S[i+1] < \cdots < S[m]$ and therefore all numbers after $S[i]$ also form inversions with $T[j]$. In other words, $(i,j)$, $(i+1,j)$, $(i+2,j)$, $\cdots$, $(m,j)$ — all these are inversions. The number of these inversions are $(m-i+1)$; hence we increase *inv* by this amount. Note that also at this time point, $T[j]$ is copied/moved to $C$ and $j$ advances. $T[j]$ won't have a chance to be explicitly compared with any of the $S[i+1], \cdots, S[m]$. These comparisons are saved, as they are guaranteed to be forming inversions with $T[j]$, and therefore counted right here (when $S[i] > T[j]$ happens).

More precisely, when $S[i] > T[j]$ happens, $S[i]$ must be the smallest number in $S$ that is larger than $T[j]$. In other words, it is not possible to have $i'$ such that $i' < i$ and $S[i'] > T[j]$. Let's prove this by contradiction. Assume that such $i'$ exists. When $S[i] > T[j]$ happens, $S[i']$ must have been added to $C$ simply because $i' < i$. Also right after $S[i] > T[j]$ happens, $T[j]$ will be added $C$. This implies that $S[i']$ is added to $C$ before $T[j]$. This is a contradiction because $S[i'] > T[j]$ but we proved (in Lecture 01) that $C$ will be sorted (in ascending order). The claim that, when $S[i] > T[j]$ happens $S[i]$ is the smallest number in $S$ that is larger than $T[j]$, implies that there are *exactly* $m-i+1$ inversions involving $T[j]$; all numbers before $S[i]$ do not form inversions with $T[j]$ because they are smaller than $T[j]$.

Hence, above algorithm does this: when $T[1]$ is added to $C$, it counts exactly all inversions that involves $T[1]$, i.e., inversions in the form of $(\cdot, 1)$; when $T[2]$ is added to $C$, it counts exactly all inversions that involves $T[2]$, i.e., inversions in the form of $(\cdot, 2)$; and so on. Since each number in $T$ will be added once and exactly once to $C$, eventually all inversions are precisely counted.

Here is a working example: $S = [2,4,5,7], T = [1,3,6,9]$. Let's run above algorithm to verify these claims. $k = 1, i = 1, j = 1, S[1] > T[1]$, so $C = [1]$ and inv = 4; 4 is the #inversions $(\cdot, j = 1)$

$k = 2, i = 1, j = 2, S[1] < T[2]$, so $C = [1,2]$;
$k = 3, i = 2, j = 2, S[2] > T[2]$, so $C = [1,2,3]$ and inv = 4 + 3; 3 is the #inversions $(\cdot, j = 2)$
$k = 4, i = 2, j = 3, S[2] < T[3]$, so $C = [1,2,3,4]$;
$k = 5, i = 3, j = 3, S[3] < T[3]$, so $C = [1,2,3,4,5]$;
$k = 6, i = 4, j = 3, S[4] > T[3]$, so $C = [1,2,3,4,5,6]$ and inv = 4 + 3 + 1; 1 is the #inversions $(\cdot, j = 3)$
$k = 7, i = 4, j = 4, S[4] < T[4]$, so $C = [1,2,3,4,5,6,7]$;
$k = 8, i = 5, j = 4, S[5] > T[4]$, so $C = [1,2,3,4,5,6,7,9]$ and inv = 4 + 3 + 1 + 0. 0 is the #inversions $(\cdot, j = 4)$

The running time of merge-count-inv-two-sorted-arrays is clearly $\Theta(m+n)$, the same to the merge-two-sorted-arrays function. Why is it faster than count-inv-two-arrays (both functions count the number of inversions between two arrays)? The merge-count-inv-two-sorted-arrays function successfully avoid the all-vs-all comparisons, by taking advantage of the fact that $S$ and $T$ are sorted. (Again, key: when $S[i] > T[j]$ happens, $S[i+1], S[i+2], \cdots, S[m]$ are all guaranteed to be forming inversions with $T[j]$ so these comparisons are avoided.)

Let's analyze the running time of DC-sort-count-inv. Since merge-count-inv-two-sorted-arrays $(A'_1, A'_2)$ takes $\Theta(|A'_1| + |A'_2|) = \Theta(n)$ time, the total running time of DC-sort-count-inv, denoted as $T(n)$, can be written as $T(n) = 2T(n/2) + \Theta(n)$. This gives $T(n) = \Theta(n \log n)$.

# Lecture 5 Selection Problem

The *selection* problem is formally defined as follows: given an array $A[1\cdots n]$ and an integer $k$, $1 \le k \le n$, to find the $k$-th smallest number in $A$. Here we assume that all numbers in $A$ are distinct. The following algorithm we design can be easily extended to allow duplicated numbers in $A$.

A straightforward algorithm is sorting $A$; then the $k$-th element of the sorted array is exactly the $k$-th smallest number of $A$. This algorithm runs in $\Theta(n \log n)$ time. Can we do better? Notice that, when $k = 1$, this problem is to seek the smallest number in $A$; when $k = n$, this problem is to seek the largest number in $A$. In either case, we know that it can be done in linear time. In fact, the general case can also be solved in linear time, using a divide-and-conquer approach, which is the focus of this lecture. Note also that, when $k = n/2$ the selection problem seeks the *medain* of $A$.

### Pivot-based Divide-and-Conquer Algorithm

We will design a *pivot-based* divide-and-conquer algorithm. The idea is to first choose a number in $A$, called pivot, denoted as $x$. We then partition $A$, using $x$, into 3 parts, $(A_1, x, A_2)$, where $A_1$ (resp. $A_2$) stores the numbers in $A$ that are smaller (resp. larger) than $x$. Specifically, we initialize two empty lists $A_1$ and $A_2$, we then examine all numbers: for each $1 \le i \le n$, we put $A[i]$ to $A_1$ if $A[i] < x$, and we put $A[i]$ to $A_2$ if $A[i] > x$.

Now, with $A_1$ and $A_2$ in hand, we can locate which part contains the $k$-th smallest number of $A$ (and therefore discard the other two parts). Specifically, if $k \le |A_1|$, then we know that the $k$-th smallest number of $A$ must be in $A_1$, and it is exactly the $k$-th smallest number of $A_1$; if $k = |A_1| + 1$, then we know that the $k$-th smallest number of $A$ must be $x$; if $k > |A_1| + 1$, then we know that the $k$-th smallest number of $A$ must be in $A_2$, and it is exactly the $(k - |A_1| - 1)$-th smallest number of $A_2$.

*Example.* Let $A = [15, 5, 2, 20, 1, 9, 4, 13, 8]$ and $k = 7$. Suppose that we are given pivot $x = 8$. We partion $A$ into $A_1, x, A_2$, where $A_1 = [5, 2, 1, 4]$ and $A_2 = [15, 20, 9, 13]$. As $k = 7 > |A_1| + 1 = 5$, the 7th smallest number of $A$ must be the 2nd (i.e, $7 - 5$) smallest number of $A_2$.

The above analyis is illustrated with the following pseudo-code.

```
Algorithm selection (A, k)
    x = find-pivot (A);
    for i = 1 to |A|
        if A[i] < x: add A[i] to A1
        if A[i] > x: add A[i] to A2
    end
    if k ≤ |A1|: return selection (A1, k);
    else if k = |A1| + 1: return x;
    else: return selection (A2, k − |A1| − 1);
end algorithm;
```

We don't know how to find a (good) pivot yet. But no matter how we do it, the algorithm is always correct, i.e., it always find the $k$-th smallest number of $A$. The choice of $x$ only affects the running time of this algorithm, as it affects $|A_1|$ and $|A_2|$.

Let's first have a look at the running time of above algorithm to find a clue what pivot we need to target. Let $T(n)$ be the running time of selection $(A, k)$ when $|A| = n$. We note that this definition is independent

of $k$; in other words, it is the running time for the *worst* choice of $k$. We use $P(n)$ to denote the running time of find-pivot $(A,k)$ with $|A| = n$. Partitioning $A$ into $A_1, x, A_2$ clearly takes $\Theta(n)$ time. For the remaining if-else-if-else part, again we assume the *worst-case* scenario, i.e., the larger array among $A_1$ and $A_2$ will always be chosen. Combined, we have this recurrence: $T(n) = P(n) + \Theta(n) + T(\max\{|A_1|, |A_2|\})$.

## Finding a Good Pivot

A good choice of pivot should result in $A_1$ and $A_2$ as balanced as possible, as in this case $\max\{|A_1|, |A_2|\}$ will be minimized. The best case, of course, is that $x$ is the median of $A$ which gives $|A_1| = |A_2|$. However, calculating the median of $A$ is kind of as hard as solving the selection problem. Instead, we can try to pick a pivot $x$ that is close to the median of $A$, using the idea called "median of medians".

Here is the procedure. We partition $A$ into $n/5$ subarrays, each of which has a size of 5. We then calculate the median (i.e., the 3rd smallest number) of each subarray. We collect these medians with an array $M$. Clearly, $|M| = n/5$. We then calculate the *median* of $M$, which will be the pivot we will use.


$M = (m_1,\ m_2,\ m_3,\ m_4)$

Figure 1: Finding pivot $x$ using median of medians. The median of each subarray (of size 5) is marked with $m_i$. We then collect these medians and denote it as $M$. The median of $M$ will be the pivot $x$.

There are two questions here. First, how to calculate the median of each subarray? As each subarray is of size 5, we can use any algorithm. For example, we can sort it, using time of $5 \cdot \log 5$, to get its median. Note, as there are $n/5$ subarrays, the total running time will be $n/5 \cdot 5 \cdot \log 5 = \log 5 \cdot n = \Theta(n)$. Second, how to calcuate the median of $M$? The answer is a recursive call: selection $(M, |M|/2)$. We will see later on that, the resulting algorithm still runs in linear time.

The pseudo-code for find-pivot function is given below.

```
function find-pivot (A)
    if |A| < 5: find (e.g., by sorting A) and return the median of A;
    partition A into n/5 subarrays of size 5;
    calculate the median of each subarray;
    let M be the array that includes all medians;
    return selection (M, |M|/2);
end
```

## Running Time of the Selection Algorithm

By definition, the find-pivot functions takes time $\Theta(n) + T(|M|) = \Theta(n) + T(n/5)$, as $|M| = n/5$. Therefore, the total running time, in the form of a recurrence, is $T(n) = \Theta(n) + T(n/5) + \max\{T(|A_1|), T(|A_2|)\}$.

We now bound $\max\{|A_1|, |A_2|\}$. Think: how many numbers in $A$ are *guaranteed* smaller than $x$ (this number then gives a lower bound of $|A_1|$)? First, in these $n/5$ medians, half of them, i.e., $n/10$ numbers, are smaller than $x$, as $x$ is the median of these medians. Second, consider these $n/10$ subarrays whose median is smaller than $x$: clearly in each of these subarrays, at least two numbers are smaller than $x$. This is because the median of this subarray (with 5 numbers) is smaller than $x$ and there are two numbers in this subarray that are smaller than its median. Combined, we have a lower bound of $|A_1|$: $|A_1| \geq n/10 + 2 \cdot n/10 = 3n/10$.

This gives an upper bound of $|A_2|$: $|A_2| = n - |A_1| \leq 7n/10$.

Symmetrically, we have that $|A_2| \geq 3n/10$ and hence $|A_1| = n - |A_2| \leq 7n/10$. This is because, in these $n/5$ medians, $n/10$ of them are larger than $x$, and in these corresponding $n/10$ subarrays whose median is larger than $x$, there are in total $2 \cdot n/10$ numbers larger than $x$.

Combined, we have $\max\{|A_1|, |A_2|\} \leq 7n/10$. The above recurrence becomes $T(n) \leq \Theta(n) + T(n/5) + T(7n/10)$. Note that the recurrence is $T(n) \leq \cdots$ instead of $T(n) = \cdots$, as we only have an upper bound for $\max\{|A_1|, |A_2|\}$. (In other words, we don't have that $\max\{|A_1|, |A_2|\} = 7n/10$.) How to solve this recurrence? First, we can use a conclusion stated and proved in the next subsection. Suppose the recurrence is "=", i.e., $T'(n) \leq \Theta(n) + T'(n/5) + T'(7n/10)$. Applying the conclusion given below, we have $c_1 + c_2 = 1/5 + 7/10 < 1$. Hence its running time $T'(n) = \Theta(n)$. Now come back to $T(n)$. Since $T(n) \leq T'(n)$, we know that $T(n) = O(n)$. To show that $T(n) = \Omega(n)$, we need the fact that the algorithm requires the partition step which takes $\Theta(n)$ time. Hence, $T(n) = \Omega(n)$. Combined, we have $T(n) = \Theta(n)$.

## Solving the Recurrence

Consider recurrence relation $T(n) = \Theta(n) + T(a \cdot n) + T(b \cdot n)$, $T(1) = 1$, $0 < a < 1$, $0 < b < 1$. We now prove the following:

1. $T(n) = \Theta(n)$, if $a + b < 1$.
2. $T(n) = \Theta(n \cdot \log n)$, if $a + b = 1$.

Assume that the term $\Theta(n)$ in the recursion admits a lower bound of $d_1 n$ and upper bound of $d_2 n$, for large enough $n$. That is $T(n) \geq d_1 n + T(an) + T(bn)$ and $T(n) \leq d_2 n + T(an) + T(bn)$.

1. We first prove that $T(n) = O(n)$, by induction. Assume that $T(n) \leq c_2 n$ for large enough $n$, where $c_2 = d_2/(1-a-b)$. Now we prove that $T(n+1) \leq c_2(n+1)$. We can write

   $$T(n+1) \leq d_2(n+1) + T(a(n+1)) + T(b(n+1)).$$

   We have $a(n+1) \leq n$ and $b(n+1) \leq n$ for large enough $n$ as $a < 1$ and $b < 1$. By the inductive assumption we have

   $$\begin{aligned} T(n+1) &\leq d_2(n+1) + c_2 a(n+1) + c_2 b(n+1) \\ &\leq d_2(n+1) + c_2(a+b)(n+1) \\ &= (d_2 + c_2(a+b))(n+1) \\ &= c_2(n+1). \end{aligned}$$

   It's easy to verify the last equation, i.e., $d_2 + c_2(a+b) = c_2$ with $c_2 = d_2/(1-a-b)$. In fact this is where we determine the value of $c_2$. You can also see why $a + b < 1$ is required here. We then prove that $T(n) = \Omega(n)$, by induction. Assume that $T(n) \geq c_1 n$ for large enough $n$, where $c_1 = d_1/(1-a-b)$. Now we prove that $T(n+1) \geq c_1(n+1)$. We can write

   $$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1) + c_1 b(n+1) \\ &\geq d_1(n+1) + c_1(a+b)(n+1) \\ &= (d_1 + c_1(a+b))(n+1) \\ &= c_1(n+1). \end{aligned}$$

   The last equation holds as $d_1 + c_1(a+b) = c_1$ with $c_1 = d_1/(1-a-b)$.

2. We first prove that $T(n) = O(n\log n)$, by induction. Assume that $T(n) \leq c_2 n\log n$ for large enough $n$, where $c_2 = -d_2/(a\log a + b\log b)$. Now we prove that $T(n+1) \leq c_2(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\leq d_2(n+1) + c_2 a(n+1)\log(a(n+1)) + c_2 b(n+1)\log(b(n+1)) \\ &\leq d_2(n+1) + c_2(a\log a + b\log b)(n+1) + c_2(a+b)(n+1)\log(n+1) \\ &= (d_2 + c_2(a\log a + b\log b))(n+1) + c_2(n+1)\log(n+1) \\ &= c_2(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_2 + c_2(a\log a + b\log b) = 0$ with $c_2 = -d_2/(a\log a + b\log b)$. We then prove that $T(n) = \Omega(n\log n)$, by induction. Assume that $T(n) \geq c_1 n\log n$ for large enough $n$, where $c_1 = -d_1/(a\log a + b\log b)$. Now we prove that $T(n+1) \geq c_1(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1)\log(a(n+1)) + c_1 b(n+1)\log(b(n+1)) \\ &\geq d_1(n+1) + c_1(a\log a + b\log b)(n+1) + c_1(a+b)(n+1)\log(n+1) \\ &= (d_1 + c_1(a\log a + b\log b))(n+1) + c_1(n+1)\log(n+1) \\ &= c_1(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_1 + c_1(a\log a + b\log b) = 0$ with $c_1 = -d_1/(a\log a + b\log b)$.

## Choices of the Size of Subarrays

Why partitioning $A$ into subarrays of size 5? Does other size work (i.e., leading to a $\Theta(n)$ algorithm)? Let's consider being of size 3. Note, in this case the algorithm is still correct. But will the algorithm still run in linear time? Let's analyze it. Now we have $|M| = n/3$, as the number of subarrays is $n/3$. In these $n/3$ medians, half of them, i.e., $n/6$ numbers, are smaller than $x$, and in these corresponding $n/6$ subarrays whose median is smaller than $x$, there are in total $1 \cdot n/6$ numbers smaller than $x$. This gives that $|A_1| \geq n/6 + n/6 = n/3$, which gives $|A_2| \leq 2n/3$. Symmetrically we can prove $|A_1| \leq 2n/3$ and combined we have $\max\{|A_1|, |A_2|\} \leq 2n/3$. The recursion in this case, will be $T(n) = \Theta(n) + T(n/3) + T(2n/3)$. In fact, now $T(n) = \Theta(n\log n)$ as $1/3 + 2/3 = 1$. In sum, choosing subarrays of size 3 won't give a linear time algorithm. (Note: by using the idea of "median-of-median-of-medians", a linear-time algorithm can still be obtained in this case; see assignment.)

How about we partition $A$ into subarrays of size 7? Now we have $|M| = n/7$, as the number of subarrays is $n/7$. In these $n/7$ medians, half of them, i.e., $n/14$ numbers, are smaller than $x$, and in these corresponding $n/7$ subarrays whose median is smaller than $x$, there are in total $3 \cdot n/14$ numbers smaller than $x$. This gives that $|A_1| \geq n/14 + 3n/14 = 2n/7$, which gives $|A_2| \leq 5n/7$. Symmetrically we can prove $|A_1| \leq 5n/7$ and combined we have $\max\{|A_1|, |A_2|\} \leq 5n/7$. The recursion in this case, will be $T(n) = \Theta(n) + T(n/7) + T(5n/7)$. So $T(n) = \Theta(n)$ as $1/7 + 5/7 < 1$. In fact, any odd size that is larger than 5 will lead to a linear-time algorithm. But bigger size will result in bigger factor in sorting these subarrays. For example, compare size of 7 and size of 5: it takes $n/7 \cdot 7 \cdot \log 7 = \log 7 \cdot n$ time to sort in the case of size 7, which is larger than $\log 5 \cdot n$ in the case of size 5.

## A Randomized Algorithm for Selection Problem

We now design a *randomized algorithm* for selection problem. The idea is simply pick the pivot uniformly at random from $A$. The pseudo-code is given below.

function find-pivot-random ($A$)
    pick pivot $x$ uniformly at random from $A$;
end function ;

First, note that the selection algorithm combined with above random function to pick pivot is correct, i.e., it will still find the $k$-th smallest number of $A$. We now analyze its running time. Again, let $T(n)$ be the running time of selection ($A$, $k$), with above random function to select pivot, when $|A| = n$. Define random variable $Z := \max\{|A_1|, |A_2|\}$. Hence we can write $T(n) = \Theta(n) + T(Z)$. Again, here $Z$ is a random variable, $T(Z)$ is a random variable, and therefore $T(n)$ is also a random variable.

We aim to calculate the expected running time, a common practice in analyzing randomized algorithms. We first estimate the distribution of $Z$. Think: what's the probability for event $Z \leq 3n/4$? Answer: at least $1/2$. Why? This is because we pick $x$ uniformly at random from $x$. Therefore, the probability of event of $\{x$ is between 25-percentile and 75-percentile of $A\}$ is $1/2$. And this event is equivalent to the event that $Z \leq 3n/4$, according to the definition of $Z$. Hence, $\Pr(Z \leq 3n/4) = 1/2$.

We now calculate its expected running time. We start with recursion $T(n) = \Theta(n) + T(Z)$. We first take expectation over $Z$ on both sides: $\mathcal{E}_Z[T(n)] = \Theta(n) + \mathcal{E}_Z[T(Z)]$. Note that $T(n)$ does not contain $Z$ (although $T(n)$ is a random variable), we have $\mathcal{E}_Z[T(n)] = T(n)$. That is $T(n) = \Theta(n) + \mathcal{E}_Z[T(Z)]$.

We now estimate $\mathcal{E}_Z[T(Z)]$.

$$\begin{aligned}
\mathcal{E}_Z[T(Z)] &= \sum_{k=n/2}^{n} \Pr(Z=k)\cdot T(k) \\
&= \sum_{k=n/2}^{3n/4} \Pr(Z=k)\cdot T(k) + \sum_{k=3n/4}^{n} \Pr(Z=k)\cdot T(k) \\
&\leq T(3n/4)\cdot \sum_{k=n/2}^{3n/4} \Pr(Z=k) + T(n)\cdot \sum_{k=3n/4}^{n} \Pr(Z=k) \\
&= T(3n/4)\cdot \Pr(Z \leq 3n/4) + T(n)\cdot \Pr(Z \geq 3n/4) \\
&\leq T(3n/4)\cdot 1/2 + T(n)\cdot 1/2.
\end{aligned}$$

Hence, now we have $T(n) \leq \Theta(n) + T(3n/4)/2 + T(n)/2$, which gives $T(n) \leq \Theta(n) + T(3n/4)$. We now take expectation, over $T(n)$, on both sides: $\mathcal{E}_T[T(n)] = \Theta(n) + \mathcal{E}_T[T(3n/4)]$. By using master's theorem, we have that $\mathcal{E}_T[T(n)] = \Theta(n)$.

# Lecture 6 Closest Pair Problem

Given a set of $n$ points in 2D plane, $P = \{p_1, p_2, \cdots, p_n\}$, where each point $p_i$ is represented as its $x$-coordinate $p_i.x$ and $y$-coordinate $p_i.y$, the *closest pair* problem seeks a pair of points $p_i, p_j \in P$ such that their distance, defined as $dist(p_i, p_j) = \sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2}$, is minimized.

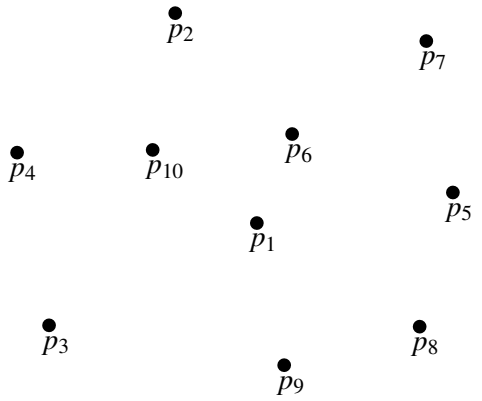



Figure 1: A set of 2D points $P = \{p_1, p_2, \cdots, p_9, p_{10}\}$. The closest pair is $(p_1, p_6)$.

A brute-force algorithm enumerates all pairs, compute their distances, and find the smallest one:

```
Algorithm #1 (P[1···n])
    use d* to store the current minimum distance, initialized as a large number M
    use (p*, q*) to store the corresponding optimal pair, initialized as NULL
    for i = 1 to n
        for j = i + 1 to n
            d = dist(P[i], P[j]);
            if d < d*: d* = d, p* = P[i], q* = P[j];
        end
    end
end algorithm;
```

The above algorithm certainly takes $\Theta(n^2)$ time. We aim for designing a faster, $\Theta(n \cdot \log n)$ algorithm. To achieve this, we certainly need to avoid the above all-vs-all comparisons. How? Let's consider the 1D closest pair problem, i.e., the given $n$ points $P = \{p_1, p_2, \cdots, p_n\}$ all locate on the $x$-axis (now each point is represented by a single coordinate rather than two coordinates). See the example below. To find the closest pair, we can sort all points from left to right. Let $P' = (p'_1, p'_2, \cdots, p'_n)$ be the sorted list of these 1D points. It is obvious that the closest pair of $P$ must be adjacent in $P'$, i.e., the closest pair must be $(p'_i, p'_{i+1})$ for some $i$. Hence, we can compute $dist(p'_i, p'_{i+1})$ for all $1 \le i \le n-1$ and pick the minimum. The whole process takes $\Theta(n \log n)$ time as the sorting step takes $\Theta(n \log n)$ time and the following step takes $\Theta(n)$ time.

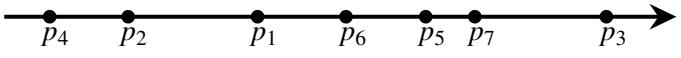



Figure 2: A set of 1D points $P = \{p_1, p_2, \cdots, p_7\}$. After sorting we get $P' = (p_4, p_2, p_1, p_6, p_5, p_7, p_3)$, and the closest pair must be next to each other in $P'$; in this case it is $(p_5, p_7)$.

For 1D closest pair problem, after sorting one point $p_i$ only needs to be compared with one other point (i.e.,

its successor in the sorted list $P'$). We will try to borrow these ideas to the 2D case. That is, we will try to sort the points (in some way), and we hope to show that it suffices to compare one point to constant number of other points. Such extension to 2D case is by no means obvious though.

We now design a divide-and-conquer algorithm, aiming for achieving a time complexity of $\Theta(n \log n)$. In a preprocessing step, we sort all points in $P$ according to their $x$-coordinates in ascending order, obtaining a sorted list denoted as $P_X$. We will also sort all points according to their $y$-coordinates in ascending order, obtaining a sorted list denoted as $P_Y$. For example in Figure 1, $P_X = (p_4, p_3, p_{10}, p_2, p_1, p_9, p_6, p_8, p_7, p_5)$, and $P_Y = (p_9, p_8, p_3, p_1, p_5, p_4, p_{10}, p_6, p_7, p_2)$. You will see later, that we need both sorted lists.

Since we design a divide-and-conquer algorithm, we need to define the recursive function. We define DC-closest-pair $(P_X, P_Y)$, where $P_X$ and $P_Y$ are the sorted lists of the same set of points $P$ w.r.t. $x$- and $y$-coordinates, returns the closest pair $(p^*, q^*)$ in $P$. We follow the framework of divide-and-conquer, that partitions the input into smaller subproblems. Naturally, we can partition the points in $P$ into two halves, $L$ and $R$, based on their $x$-coordinates. For example in Figure 1, we will have $L = \{p_1, p_2, p_3, p_4, p_{10}\}$ and $R = \{p_5, p_6, p_7, p_8, p_9\}$. Note that, in order to recursively call DC-closest-pair on $L$ and $R$, we have to create the corresponding sorted lists for both $L$ and $R$. That is, for the left partition $L$, we have to create $L_X$ and $L_Y$, the sorted lists of $L$ w.r.t. $x$- and $y$-coordinates, and for $R$ we need to create $R_X$ and $R_Y$. In the algorithm we will create lists $L_X, L_Y, R_X, R_Y$; we will not explicitly create the two halves $L$ and $R$.

Since $P_X$ is given, and $L$ and $R$ are partitioned based on $x$-coordinates, the construction and $L_X$ and $R_X$ is easy: $L_X$ is simply the first half of $P_X$, and $R_X$ is the remaining half, i.e., $L_X = P_X[1, 2, \cdots, n/2]$, and $R_X = P_X[n/2+1, \cdots, n]$. The construction of $L_Y$ (resp. $R_Y$) can be done by sorting the points in $L_X$ (resp. $R_X$), but this will take $\Theta(n \cdot \log n)$ time, which will make the final algorithm run in $\Theta(n \cdot \log^2 n)$ time. In fact, the construction of $L_Y$ and $R_Y$ can be done in $\Theta(n)$ time, by making use of the given $P_Y$, which sorts all points (i.e., $P$) by their $y$-coordinates. We can tranverse $P_Y$, and for each point $p$, put it to the end of $L_Y$ if $p$ locates in $L$, and put it to the end of $R_Y$ if $p$ locates in $R$. Since we keep the order of $P_Y$, certainly the resulting $L_Y$ and $R_Y$ are sorted by $y$-coordinates. The remaining question is how to determine $p$ locates in which side. This can be determined by comparing the $x$-cooridnate of $p$, i.e., $p.x$, with the $x$-coordinate of the rightmost point in $L$, i.e., $L_X[n/2].x$. If $p.x \leq L_X[n/2].x$, then $p$ is in $L$ and otherwise $p$ is in $R$. For the example in Figure 1, $L_X = (p_4, p_3, p_{10}, p_2, p_1)$, $R_X = (p_9, p_6, p_8, p_7, p_5)$, $L_Y = (p_3, p_1, p_4, p_{10}, p_2)$, and $R_Y = (p_9, p_8, p_5, p_6, p_7)$.

Now we recursively call DC-closest-pair $(L_X, L_Y)$ and DC-closest-pair $(R_X, R_Y)$, which return the closest pair in $L$ and the closest pair in $R$. We denote them $(p_L^*, q_L^*)$ and $(p_R^*, q_R^*)$, respectively. We calculate the distances between them, and calculate the smaller of the two: $d^* := \min\{dist(p_L^*, q_L^*), dist(p_R^*, q_R^*)\}$. Clearly, $d^*$ is the closest pair we find so far. But it may not the closest pair in $P$, since we have not considered pairs that span the two sides. We cannot afford to compare all pairs between $L$ and $R$, as it takes $\Theta(n^2)$ time. We need a more efficient way to do it. The critical information is that we already have a pair with distance $d^*$, and therefore we only need to determine if there exists any pair (spanning the two sides) that is closer than $d^*$. Although this argument is straightforward, it is actually very powerful to avoid $\Theta(n^2)$ of comparisons.

Following above argument, we only need to consider points within a band centered at the partition line, i.e., $x = L_X[2/n].x$. See Figure 3. The width of this band is $2 \cdot d^*$. Any point outside this band must have a distance larger than $d^*$ with any point on the other side. Hence, if a pair spanning $L$ and $R$ have a distance smaller than $d^*$, then both points must be in the band.

So we narrowed down the points to those in the band. But we still cannot afford comparing all pairs in it. Fortunately, we do not have to. In fact, for each point in the band, we only *need* to compare it to a *constant* number of points. This boils down the total number of comparisons to $\Theta(n)$. Let's establish this key result. First, we need to create a sorted list for points in the band, in ascending order of $y$-coordinates.

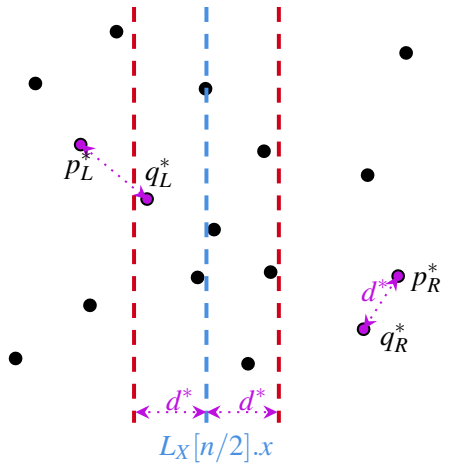



Figure 3: Illustration of the band centered at line $x = L_X[n/2].x$ with width $2 \cdot d^*$.

We denote this sorted list as $B_Y$. The construction of $B_Y$, again, can be done by using the given sorted list $P_Y$. Specifically, we tranverse $P_Y$, and for each point $p$ in it, determine if it is in the band, by testing if $|p.x - L_X[n/2].x| \leq d^*$, and if it is so, we put $p$ to the end of $B_Y$. The lemma given below states the key conclusion: if two points are 12 positions or more apart in the sorted list $B_Y$, then their distance is guaranteed to be larger than $d^*$. With this lemma, in order to determine possible pair with distance smaller than $d^*$ in the band, we just need to examine pairs $(B_Y[i], B_Y[j])$, where $j \leq i + 12$. The number of such pairs is certainly at most $12 \cdot |B_Y|$, which is $\Theta(n)$.

*Lemma.* If $j \geq i + 12$, then $dist(B_Y[i], B_Y[j]) \geq d^*$.

We now prove above lemma. We partition the band into squares of size $d^*/2 \times d^*/2$. See Figure 4. We first prove that, there is at most 1 point in each box (including its boundary). Suppose conversely that there are two points, $p$ and $q$, in a box. Then the distance between them $dist(p,q) \leq d^*/2 \cdot \sqrt{2} < d^*$. This is a contradiction, since each box is entirely inside either left side or the right side, and we know that the closest pair not spanning two sides is $d^*$. Next, if $j \geq i + 12$, then $B_Y[j]$ is at least three layers of boxes above $B_Y[i]$. This is because, each layer contains 4 boxes, and therefore at most 4 points can be in them. Hence if $B_Y[i]$ is in layer-$l$, then $B_Y[j]$ must be in layer-$(l+3)$ or higher. The height of each layer is $d^*/2$. The vertical

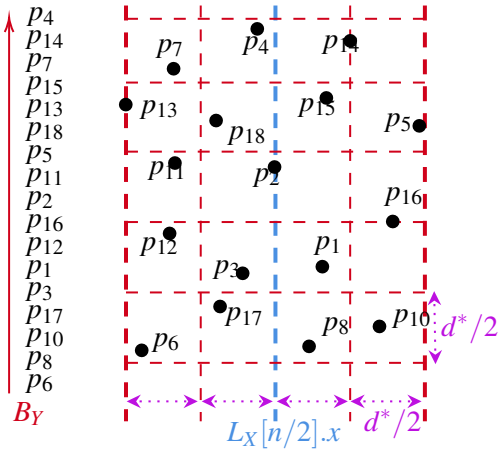



Figure 4: Illustration the proof of above lemma.

distance between $B_Y[i]$ and $B_Y[j]$ is hence larger than $d^*$; therefore, their distance $dist(B_Y[i], B_Y[j]) > d^*$.

The complete algorithm is given below in pseudo-code.

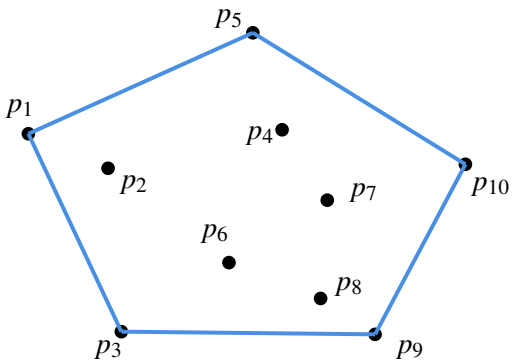
```
Algorithm #2 (P[1 ··· n])
    sort P according to x-coordinates to obtain P_X;
    sort P according to y-coordinates to obtain P_Y;
    return DC-closest-pair (P_X, P_Y);
end algorithm;

function DC-closest-pair (P_X, P_Y)
    if |P_X| ≤ 3: enumerate all pairs to find the closest pair and return it;
    L_X = P_X[1 ··· n/2], R_X = P_X[n/2 + 1 ··· n];
    init L_Y and R_Y as empty lists;
    let x* = L_X[n/2].x;
    for i = 1 to |P_Y|
        if P_Y[i].x ≤ x*: add P_Y[i] to the end of L_Y;
        else: add P_Y[i] to the end of R_Y;
    end
    (p*_L, q*_L) = DC-closest-pair (L_X, L_Y);
    (p*_R, q*_R) = DC-closest-pair (R_X, R_Y);
    d* = min{dist(p*_L, q*_L), dist(p*_R, q*_R)};
    (p*, q*) = arg min{dist(p*_L, q*_L), dist(p*_R, q*_R)}, to store the closest pair found from two halves;
    init B_Y as an empty list;
    for i = 1 to |P_Y|
        if |P_Y[i].x − x*| ≤ d*: add P_Y[i] to the end of B_Y;
    end
    for i = 1 to |B_Y|
        for j = i + 1 to i + 12
            d = dist(B_Y[i], B_Y[j]);
            if d < d*: d* = d, p* = B_X[i], q* = B_Y[j];
        end
    end
    return (p*, q*);
end algorithm;
```

Let's analyze the time complexity. Let $T(n)$ be running time of DC-closest-pair ($P_X$, $P_Y$) when $|P_X| = |P_Y| = n$. In DC-closest-pair function, the two recursive calls takes $2 \cdot T(n/2)$ time; the rest takes $\Theta(n)$ time. Therefore, the recurrence is $T(n) = 2 \cdot T(n/2) + \Theta(n)$ which gives $T(n) = \Theta(n \cdot \log n)$. The entire algorithm #2 is therefore also takes $\Theta(n \cdot \log n)$ time since the two sorts takes $\Theta(n \cdot \log n)$ as well.

# Lecture 7 Convex Hull

**Definition 1** (Convex Hull)**.** Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of points on 2D plane, each of which is represented as $p_i = (x_i, y_i)$. The *convex hull* of $P$, denoted as $CH(P)$, is defined as the smallest, convex polygon that includes all points in $P$.

A convex hull is usually represented as the list of vertices (which is subset of $P$) of the polygon in counter-clockwise order. Such list is circular, and it's equivalent to shifting any number of vertices. If a point $p \in P$ is one of the vertices of the convex hull of $P$, we usually say that $p$ is *on* the convex hull, and denote it as $p \in CH(P)$.




$$CH(P) = (p_1,\ p_3,\ p_9,\ p_{10},\ p_5)$$

Figure 1: A set of points $P = \{p_1, p_2, \cdots, p_{10}\}$ and its convex hull.

For most computational geometry problems, we always assume that the data (points, lines, etc) given to us are in *general* situation. For example, we usually assume no three (or more) lines go through the same point, no three (or more) points are colinear, and no four points are on the same circle, etc.

**Property 1.** Let $p \in P$. Then $p \in CH(P)$ if and only if there exists a line $l$ such that $p$ is on $l$, and that all other points, i.e. $P \setminus \{p\}$, are on the same side of $l$.

The above property gives us a way to determine if a point is on the convex hull, which will be used in the following algorithm (e.g., Graham-Scan).

**Property 2.** For any set of points $P$ and another point $q$, we have $CH(P \cup \{q\}) = CH(CH(P) \cup \{q\})$.

The above property gives a way to gradually contruct convex hull. Suppose we already know the convex hull of $P$, i.e., $CH(P)$. Now we want to calculate the convex hull of $P \cup \{q\}$. We only need to consider these points in $CH(P)$ and $q$: those points that are inside of the convex hull of $P$ can be safely discarded.

## Graham-Scan Algorithm

The idea of Graham-Scan is to gradually construct the convex hull. The first step of this algorithm is the identification of the lowest point in $P$, i.e., the point with smallest $y$-coordinate, denoted as $p_*$. The second step is to sort all others points in counter-clockwise order w.r.t. $p_*$. Specifically, for any point $p \in P$, the measure we use in sorting is the *angle* defined by points $p$, $p_*$ and $(+\infty, y\text{-coordinate of } p_*)$. We define such angle for $p_*$ is 0. All points are then sorted in ascending order by their angles. After sorting, for the sake of simplicity, we rename all points in $P$ in this order, i.e., rename $p_*$ as $p_1$, the point with second smallest angle as $p_2$, and so on. These two steps can be regarded as *preprocessing* steps of the Graham-Scan algorithm (i.e., first 3 lines of Algorithm Graham-Scan); the resulting of it is a sorted list of points $P$. See Figure 2.

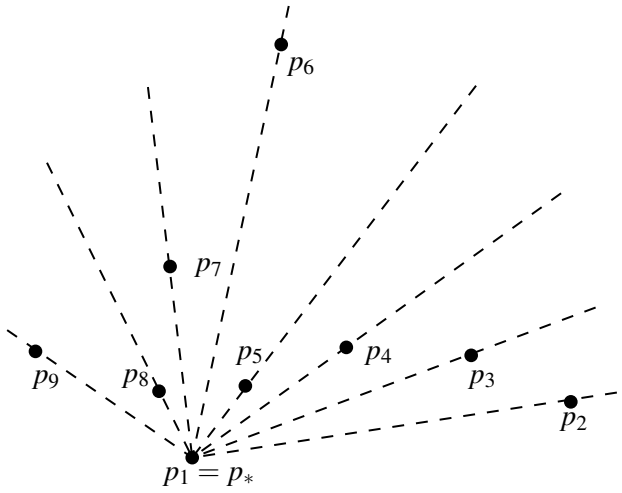



Figure 2: Preprocessing steps of Graham-Scan.

The main body of Graham-Scan is referred to as Graham-Scan-Core. It processess points in $P$ (notice that now they are properly sorted) one by one and gradually construct the convex hull. It maintains a *stack* data structure, called $S$, and keeps the following variant:

*Invariant:* after processing the first $k$ points of $P$, the convex hull of these $k$ points, i.e., $CH(p_1, p_2, \cdots, p_k)$ will be stored in the stack $S$: the list of points in $S$ from bottom to top gives the list of the vertices of the convex hull in counter-clockwise order.

How to guarantee above invariant? Consider the general case when we are processing $p_k$. Now we know that $S$ stores $CH(p_1, p_2, \cdots, p_{k-1})$, and the goal is to determine $CH(p_1, p_2, \cdots, p_k)$. Following above Property 2, we know that $CH(p_1, p_2, \cdots, p_k) = CH(CH(p_1, p_2, \cdots, p_{k-1}) \cup \{p_k\})$. Hence we only need to consider the points in $S$ and the current point $p_k$.

But the issue is that, points in $S$, which is exactly $CH(p_1, p_2, \cdots, p_{k-1}$, may not be in $CH(p_1, p_2, \cdots, p_k)$. We therefore need to *determine* and *exclude* such points in $S$. How? See Figure 3 for an example. We use the following procedure to determine if the top element, called $p_a$, of $S$ is on $CH(p_1, p_2, \cdots, p_k)$: we check the *orientation* of the three points $p_b \to p_a \to p_k$, where $p_a$ and $p_b$ are the top and second top elements of $S$ respectively. If it's "turning right", then $p_a$ will not be on $CH(p_1, p_2, \cdots, p_k)$. Why? Because in this case $p_a$ will be within triangle $p_1 p_b p_k$ (the angle of $p_a$ is in the middle of that of $p_b$ and $p_k$). If this happens, we can

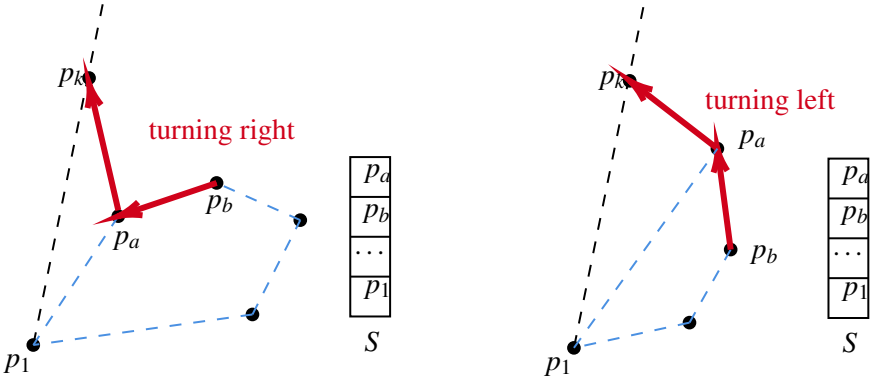



Figure 3: Procedure to decide if top element of $S$ is on $CH(p_1, p_2, \cdots, p_k)$.

remove $p_a$ safely, i.e., by calling *pop (S)*, as $p_a$ cannot be in the convex hull of the first $k$ points. After we remove $p_a$ from $S$, we will need to continue to examine the top element of $S$ iteratively. If the orientation is "turning left", then actually we get the convex hull of the first $k$ points—that's the points in $S$ plus $p_k$, where $p_k$ will be pushed into $S$ and become the top element of $S$. We will give a formal proof after the algorithm.

The complete Gramham-Scan algorithm is illustrated with pseudo-code below.

```
Algorithm Graham-Scan (P)
    calculate point p* in P with smallest y-coordinate;
    sort P in ascending order of the angles w.r.t. p* and (∞, 0);
    rename points in P so that points in P are following above order;
    Graham-Scan-Core (P);
end algorithm;

Algorithm Graham-Scan-Core (P = {p1, p2, ···, pn})
    init empty stack S;
    push (S, p1);
    push (S, p2);
    push (S, p3);
    for k = 4 to n
        while (S is not empty)
            let pa and pb be the top two elements of S;
            check the orientation of pb → pa → pk;
            if "turning right": pop (S) and then continue the while loop;
            if "turning left": break the while loop;
        end while;
        push (S, pk);
    end for;
end algorithm;
```

Now we show that, when the while-loop breaks, i.e., the orientation is "turning left", then $S \cup \{p_k\}$ becomes the convex hull of $\{p_1, p_2, \cdots, p_k\}$. The proof is based on the definition of convex-hull: we will verify that $S \cup \{p_k\}$ is the smallest, convex polygon that includes all $\{p_1, p_2, \cdots, p_k\}$. First, we verify that $S \cup \{p_k\}$ forms a convex polygon. This is because, all consecutive 3 points of $S$ are "turning left", as this is guaranteed by the algorithm: we push $p_k$ right after verifying $p_b \to p_a \to p_k$ are turning left. To verify that it is circularly "turning left", we further need to show that two more consecutive 3 points, namely, $(p_a, p_k, p_1)$ and $(p_k, p_1, p_2)$, are also turning left. These are easy to see as $p_k p_1$ are the rightmost line and all other points, including $p_1$ and $p_a$ are on the right side of this line. Second, we verify that the convex polygon formed by $S \cup \{p_k\}$ contains all points in $\{p_1, p_2, \cdots, p_k\}$. To see this, we just need to show that all poped points are not possible to be in $CH(p_1, p_2, \cdots, p_k)$, and this is true, as every poped point is inside of a triangle (Figure 3), and therefore must be inside of the convex hull. Third, we show that $S \cup \{p_k\}$ is the smallest convex polygon that include all first $k$ points. This is obvious as $S \cup \{p_k\}$ is a subset of the first $k$ points.

The first 3 lines of the algorithm corresponds to the preprocessing steps, which produce a sorted list of points. The main body of the algorithm, referred to as "Graham-Scan-Core", gradually construct the convex hull and guarantees above invariant. When it terminates, the convex hull of all vertices are stored $S$.

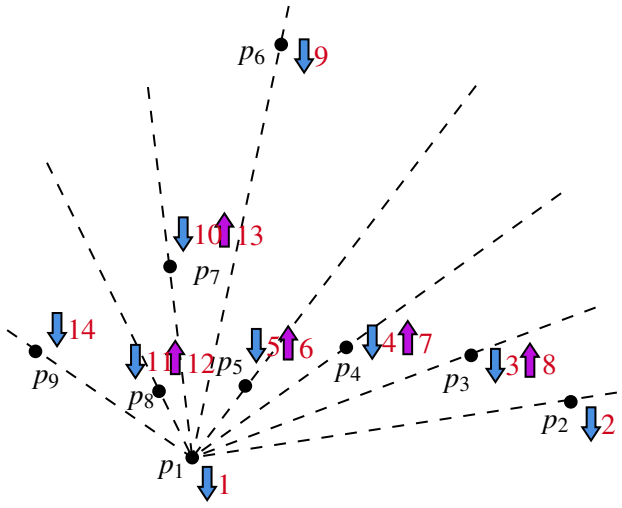



Figure 4: Running Graham-Scan-Core on the example given in Figure 2. Blue and pink arrows show the push and pop operations; the red numbers show the sequential of these operations.

Graham-Scan-Core can be regarded as an independent algorithm (you will find its use in the divide-and-conquer algorithm for convex hull). Its input is a list of points $P$, in which the first point, i.e. $p_1$, is the point with smallest $y$-coordinate, and the following points, i.e. $p_2, p_3, \cdots, p_n$, are placed in ascending order by their angle w.r.t. $p_1$ and $(+\infty, p_1.y)$. The output of Graham-Scan-Core is $CH(P)$ which will be stored in the stack $S$.

There are two technical details in implementing above algorithms. First, in the preprocessing step, for each point $p_i$ do we really need to calculate the actual angle of $\angle p_i p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$? No. Notice that the only purpose of calculating these angles is sorting these points. Therefore, only the relative order matters but not the actual angles. Hence, we can calculate $\cos(\cdot)$ of the angle and use it to sort, as $\cos(\cdot)$ is monotone over $[0,\pi]$. Clearly, $\cos(\angle p_i p_* p_\infty) = \overrightarrow{p_* p_i} \cdot (1,0)/|\overrightarrow{p_* p_i}|$.

Second, in Graham-Scan-Core, how to determine the orientation of 3 points (i.e. $p_b \to p_a \to p_k$)? We will use "right-hand-rule". Define two vectors: $\overrightarrow{A} := \overrightarrow{p_b p_a}$ and $\overrightarrow{B} = \overrightarrow{p_a p_k}$. Let $\overrightarrow{A} = (x_A, y_A)$ and $\overrightarrow{B} = (x_B, y_B)$ be their coordinates, which can be calculated from the coordinates of $p_b$, $p_a$ and $p_k$. The right-hand-rule gives that

$$\begin{cases} x_A y_B - x_B y_A > 0 & \text{if and only if} \quad p_b \to p_a \to p_k \text{ "turning left"} \\ x_A y_B - x_B y_A < 0 & \text{if and only if} \quad p_b \to p_a \to p_k \text{ "turning right"} \\ x_A y_B - x_B y_A = 0 & \text{if and only if} \quad p_b, p_a, p_k \text{ "colinear"} \end{cases}$$

## Running Time of Graham-Scan

Although Graham-Scan-Core consists of two nested loop, in fact it runs in linear time! To see this, consider the types of operations and how many each type of operation the algorithm needs to do. There are three types of operations this algorithm does: *push*, *pop*, and *checking-orientation*, and each operation takes $\Theta(1)$ time. Now let's consider the times of each type will need to execute. First, notice that $\#(push) = n$, as each point will be pushed to the stack exactly once. Second, $\#(pop) \le n$, as $\#(push) \le \#(pop)$ starting from an empty stack (a point must be pushed into the stack prior to pop). Third, notice that right after *checking-orientation* it is either a *push* or a *pop* operation (i.e., it's not possible to have two *checking-orientation* operations next to each other). This implies that $\#(checking\text{-}orientation) \le \#(push) + \#(pop) \le 2n$. Combined, we have that Graham-Scan-Core takes $\Theta(n)$ time.

The preprocessing step of Graham-Scan takes $\Theta(n \log n)$, as it's dominated by the sorting step. The entire running time of Graham-Scan therefore takes $\Theta(n \log n)$ time.

### Divide-and-Conquer Algorithm

We now design a divide-and-conquer algorithm for convex hull problem. As usual, we define recursive function, say, CHDC ($P$), which takes a set of points $P$ as input and returns $CH(P)$, represented as the list of vertices of the convex polygon in counter-clockwise order.

function CHDC ($P = \{p_1, p_2, \cdots, p_n\}$)
  if $n \leq 3$: resolve this base case and return the convex hull;
  $C_1$ = CHDC ($P[1..n/2]$);
  $C_2$ = CHDC ($P[n/2+1..n]$);
  return combine ($C_1, C_2$);
end algorithm;

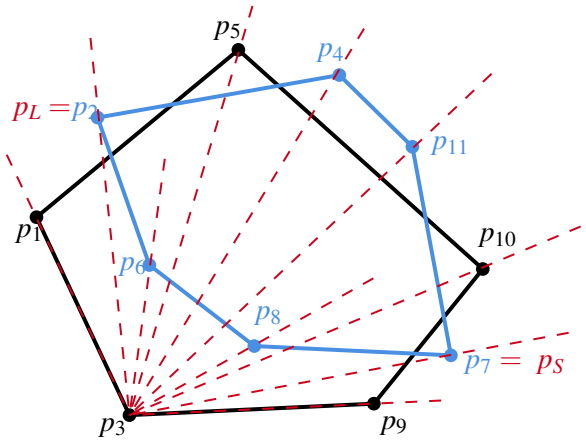
The "combine" function in above pseudo-code is missing. Before design an algorithm for "combine", we first make sure that it suffices to only consider points in $C_1$ and $C_2$. Formally, we can write $CH(P) = CH(C_1 \cup C_2)$. This can be proved using the Property 2 of convex hull. Intuitively, any point *inside* $C_1$ or $C_2$ with be inside $CH(P)$, and therefore won't be *on* $CH(P)$.

We can use, for example, Graham-Scan to calculate $CH(C_1 \cup C_2)$, which takes $\Theta(m \log m)$, where $m = |C_1 \cup C_2|$. Can we do better? Yes. We will design an $\Theta(m)$ time algorithm to calculate $CH(C_1 \cup C_2)$. The idea is to take advantage of that, $C_1$ and $C_2$ are already sorted (in counter-clockwise order along the polygon), and then to use Graham-Scan-Core to calculate $CH(C_1 \cup C_2)$. Recall that, Graham-Scan-Core is linear-time algorithm for convex hull; it just requires that all points in its input is sorted in counter-clockwise order w.r.t. the first point in its input.

The pseudo-code for combine procedure is given below. See figure below for explaining.

function combine ($C_1, C_2$)
  find $p_*$ in $C_1 \cup C_2$ with smallest $y$-coordinate;
  assume that $p_* \in C_1$; otherwise exchange $C_1$ and $C_2$;
  rewrite $C_1$ into $C_1'$ with circular shifting so that $p_*$ is the first point of $C_1'$; (*Note: now points in $C_1'$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
  find $p_S$ in $C_2$ with smallest angle (i.e., angle $\angle p_S p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$;
  find $p_L$ in $C_2$ with largest angle (i.e., angle $\angle p_L p_* p_\infty$, where $p_\infty = (+\infty, p_*.y)$;
  let $C_{2a}$ be the sublist of $C_2$ from $p_S$ to $p_L$ in counter-clockwise order; (*Note: now points in $C_{2a}$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
  let $C_{2b}$ be the sublist of $C_2$ from $p_S$ to $p_L$ in clockwise order; (*Note: now points in $C_{2b}$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
  $C_2'$ = merge-two-sorted-arrays ($C_{2a}, C_{2b}$); (*Note: merging is by the angle w.r.t. $p_*$ and $p_\infty$; after it points in $C_2'$ is sorted w.r.t. $p_*$ in counter-clockwise order.*)
  $C'$ = merge-two-sorted-arrays ($C_1', C_2'$); (*Note: merging is by the angle w.r.t. $p_*$ and $p_\infty$; after it points in $C'$, i.e., points in $C_1 \cup C_2$, is sorted w.r.t. $p_*$ in counter-clockwise order.*)
  return Graham-Core-Scan ($C'$);
end algorithm;

Clearly each step of above combine takes linear time. Therefore, the entire running time of combine is




$C_1 = (p_5, p_1, p_3, p_9, p_{10})$ $\quad C_2 = (p_4, p_2, p_6, p_8, p_7, p_{11})$

$C_1' = (p_3, p_9, p_{10}, p_5, p_1)$ $\quad C_{2a} = (p_7, p_{11}, p_4)$

$C_{2b} = (p_8, p_6, p_2)$

$C_2' = (p_7, p_8, p_{11}, p_4, p_6, p_2)$

$C' = (p_3, p_9, p_7, p_{10}, p_8, p_{11}, p_4, p_5, p_6, p_2, p_1)$

Figure 5: Example of combining $C_1$ and $C_2$.

$\Theta(|C_1|+|C_2|)$.

To obtain the running time of CHDC, we write its recursion $T(n) = 2T(n/2) + \Theta(|C_1|+|C_2|)$. In worst case, $|C_1|+|C_2| = \Theta(n)$, and therefore $T(n) = 2T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$. In this (worst) case the running time of this divide-and-conquer algorithm is the same with Graham-Scan. However, if the size of $|C_1|+|C_2|$ is in the order of $o(n)$, we will have improved running time. For example, if $|C_1|+|C_2| = \Theta(n^{0.99})$, then $T(n) = \Theta(n)$. This might happen in practical cases, although in worst case divide-and-conquer runs in $\Theta(n \log n)$ time.

# Lecture 8 Half-Plane Intersection

**Definition 1** (half-planes)**.** A line $l$ on 2D plane with function $y = ax - b$ defines two *half-planes*: the *upper half-plane*: $y \geq ax - b$ and the *lower half-plane*: $y \leq ax - b$.

**Definition 2** (upper- and lower-envelop)**.** Let $L = \{y = a_i x - b_i \mid 1 \leq i \leq n\}$ be a set of lines on 2D plane. We define the *upper-envelop* of $L$, denoted as $UE(L)$, as the intersection of the corresponding $n$ upper half-planes $\{y \geq a_i x - b \mid 1 \leq i \leq n\}$. We define the *lower-envelop* of $L$, denoted as $LE(L)$, as the intersection of the corresponding $n$ lower half-planes $\{y \leq a_i x - b \mid 1 \leq i \leq n\}$.

Either upper-envelop or lower-envelop of a set of lines can be represented as the list of lines that define its boundary from left to right. In the example below, we can write $UE(L) = (l_1, l_2, l_4, l_7)$ and $LE(L) = (l_7, l_5, l_1)$.

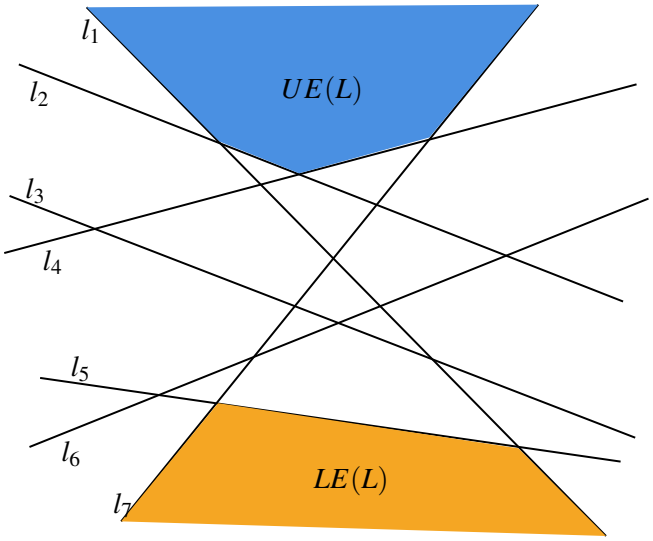



Figure 1: Illustration of upper-envelop and lower-envelop of lines $L = \{l_1, l_2, \cdots, l_7\}$.

We want to design efficient algorithms to calculate the upper- and lower-envelop of a set of lines. In fact, we don't need to design any new algorithm here. Below we will show that, the problem of finding upper- and lower-envelop of a set of lines is equivalent to the problem of finding the convex hull of a set of points. Therefore, the algorithms we've designed for finding the convex hull can be directly used to find the upper- and lower-envelop of lines.

## Duality

**Definition 3** (dual of a point)**.** Let $p = (p_x, p_y)$ be a point on 2D plane. We define the *dual* of $p$, denoted as $p^*$, as a line with function $y = p_x x - p_y$ on 2D plane.

**Definition 4** (dual of a line)**.** Let $l$ be a line with function $y = ax - b$ on 2D plane. We define its *dual*, denoted as $l^*$, as a point with coordinates $(a, b)$ on 2D plane.

The following three properties are direct consequences of above definitions. (Think how to prove them.)

**Property 1.** For any point $p$, we have $(p^*)^* = p$. For any line $l$, we have $(l^*)^* = l$.

**Property 2.** Point $p$ is on line $l$ if and only if point $l^*$ is on line $p^*$.

**Property 3.** Point $p$ is above (resp. below) line $l$ if and only if point $l^*$ is above (resp. below) line $p^*$.

## Half-plane Intersection vs. Convex Hull

**Definition 5** (upper- and lower-hull)**.** Let $P$ be a set of points, and let $CH(P)$ be the convex hull of $P$. Let $p_S \in CH(P)$ be the vertex with smallest $x$-coordinate, and $p_L \in CH(P)$ be the vertex with largest $x$-coordinate. Therefore $p_S$ and $p_L$ partition $CH(P)$ into two parts: the list of vertices from $p_S$ to $p_L$ following the counter-clockwise order is called *lower hull* of $P$, denoted as $LH(P)$; the list of vertices from $p_L$ to $p_S$ following the counter-clockwise order is called *upper hull* of $P$, denoted as $UH(P)$.

We now show that upper- and lower-envelop of lines is essentially the same with lower- and upper-hull of points. We first prove the connection between upper-envelop and lower-hull; the other one, i.e., lower-envelop and upper-hull, can be proved symmetrically.

Let $L$ be a set of lines, we define $L^* = \{l^* \mid l \in L\}$, i.e., the set of "dual points" of $L$.

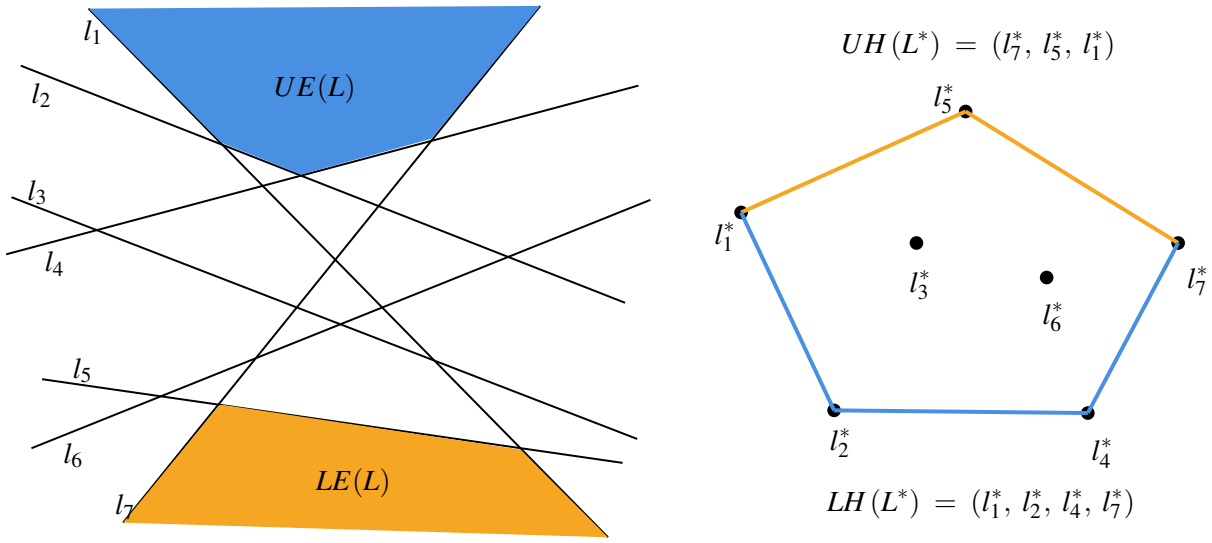



Figure 2: Illustration of duality between upper-/lower-envelop and lower-/upper-hull.

**Claim 1.** A line $l \in L$ is part of the boundary of $UE(L)$ if and only if $l^*$ is one of the vertices of $LH(L^*)$.

*Proof.* Line $l$ is part of $UE(L)$, implies that a piece of $l$ is above all other lines. This is equivalent to: there exists a point $p$, such that $p$ is on $l$, and that $p$ is above all lines in $L \setminus \{l\}$. This statement is also equivalent to the following statement, by translating everything to their dual counterparts (and applying above Properties of duality): there exists a line $p^*$, such that $l^*$ is on $p^*$ and that all points in $L^* \setminus \{l^*\}$ are above line $p^*$. Clearly, this statement is also equivalent to that $l^*$ is one vertex of the lower-hull of $L^*$ (think the Properties of convex hull). □

The above claim shows that lines in $UE(L)$ and vertices in $LH(L^*)$ are in a "dual" relationship. We now show how their ordering are connected. Recall that we represent $UE(L)$ as a list of lines from left to right. Therefore, the *slope* of these lines are in increasing order. As the dual of line $y = ax - b$ is point $(a,b)$, i.e., the slope of a line becomes the $x$-coordinate of its dual, we know that the corresponing "dual points" of $UE(L)$ are in the increasing order of their $x$-coordinates.

The above two facts can be combined as the following: $UE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $LH(L^*) = (l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. Formally, we can write

**Fact 1.** $UE(L) = (LH(L^*))^*$.

Symmetrically, with the same reasonging, we can prove that $LE(L) = (l_{p_1}, l_{p_2}, \cdots, l_{p_k})$ if and only if $UE(L^*) =$

$(l^*_{p_1}, l^*_{p_2}, \cdots, l^*_{p_k})$. (Recall that $LE(L)$ is represented as the list of lines from left to right, i.e., their slopes are decreasing, while $UH(L^*)$ is represented as the list of vertices from rightmost vertex to leftmost vertex in couter-clockwise order, i.e., their $x$-coordinates are also decreasing.) Formally, we can also write

**Fact 2.** $LE(L) = (UH(L^*))^*$.

# Lecture 9 Graph: Definitions and Representations

A graph is usually denoted as $G = (V, E)$, where $V$ represents vertices, and $E$ represents edges. There are two types of graphs, directed ones and undirected ones (see Figures below).

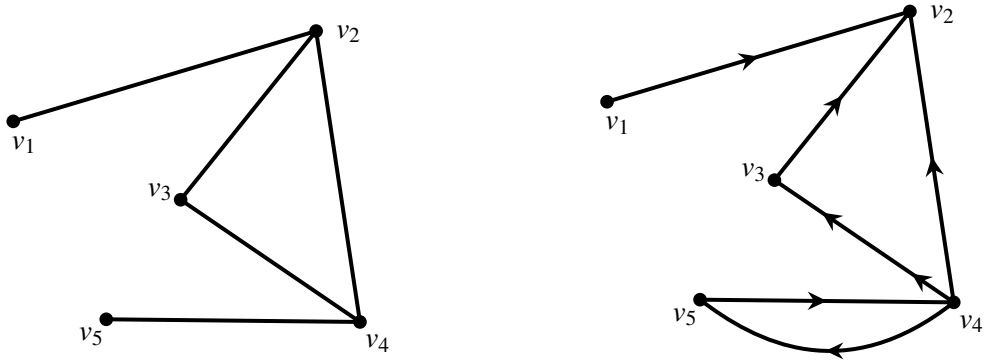
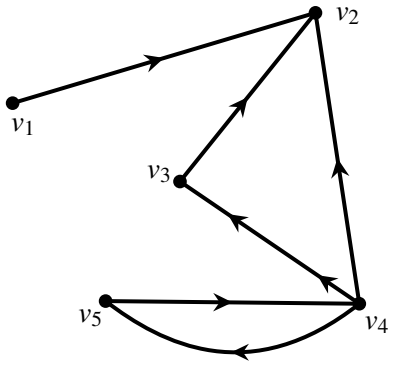



Figure 1: Left: a undirected graph $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1, v_2), (v_3, v_2), (v_4, v_3), (v_4, v_2), (v_4, v_5)\}$. Right: a directed graph $G = (V, E)$, where $V = \{v_1, v_2, v_3, v_4, v_5\}$, and $E = \{(v_1, v_2), (v_3, v_2), (v_4, v_3), (v_4, v_2), (v_4, v_5), (v_5, v_4)\}$.

Notice that for an edge $(v_i, v_j)$ in an undirected graph, the order of $v_i$ and $v_j$ are interchangable, i.e., $(v_i, v_j) = (v_j, v_i)$. In directed graph this is not the case.

Adjacency matrix and adjacency list are two commonly-used data structures to represent a graph. Adjacency matrix uses a binary matrix $M$ of size $|V| \times |V|$ to store a graph $G = (V, E)$: $M[i, j] = 1$ if and only if $(v_i, v_j) \in E$. This definition applies to both directed graphs and undirected graphs.

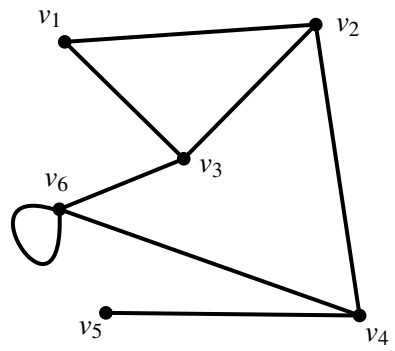
For undirected graphs, clearly the adjacency matrix $M$ is symmetric. If we assume that there is no "self-loop" edges in the form of $(v_i, v_i)$, then the number of "1"s in $M$ is exactly $2|E|$ for undirected graph. The number of "1"s in $M$ is exactly $|E|$ for directed graph.




| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
|---|---|---|---|---|---|
| $v_1$ | 0 | 1 | 0 | 0 | 0 |
| $v_2$ | 0 | 0 | 0 | 0 | 0 |
| $v_3$ | 0 | 1 | 0 | 0 | 0 |
| $v_4$ | 0 | 1 | 1 | 0 | 1 |
| $v_5$ | 0 | 0 | 0 | 1 | 0 |

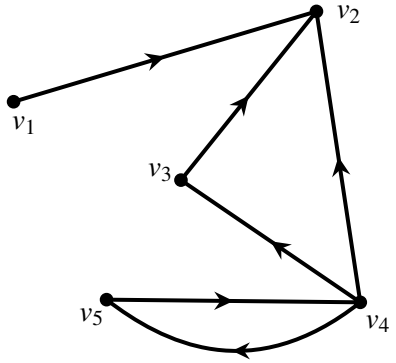
Figure 2: Adjacency matrix representation (directed graph).

Adjacency list maintains a list/array $A_i$ for each vertex $v_i \in V$, where $A_i$ stores $\{v_j \in V \mid (v_i, v_j) \in E\}$, i.e., the adjacent edges/vertices of $v_i$. A pointer is usually maintained for each vertex $v_i$ that points to the array $A_i$. And we can use an array of size $|V|$ to store these pointers. Clearly, for undirected graph, $\sum_{v_i \in V} |A_i| = 2|E|$, assuming that there is no "self-loop" edges. For directed graph, $\sum_{v_i \in V} |A_i| = |E|$.




| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ |
|---|---|---|---|---|---|---|
| $v_1$ | 0 | 1 | 1 | 0 | 0 | 0 |
| $v_2$ | 1 | 0 | 1 | 1 | 0 | 0 |
| $v_3$ | 1 | 1 | 0 | 0 | 0 | 1 |
| $v_4$ | 0 | 1 | 0 | 0 | 1 | 1 |
| $v_5$ | 0 | 0 | 0 | 1 | 0 | 0 |
| $v_6$ | 0 | 0 | 1 | 0 | 0 | 1 |

Figure 3: Adjacency matrix representation (undirected graph).




$v_1 \rightarrow (v_2)$

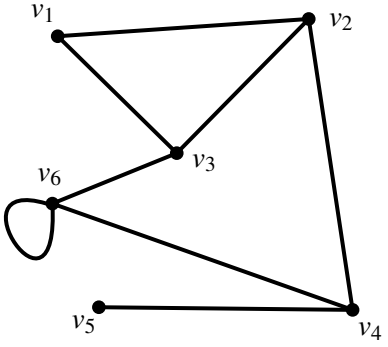
$v_2 \rightarrow ()$

$v_3 \rightarrow (v_2)$

$v_4 \rightarrow (v_2,\ v_3,\ v_5)$

$v_5 \rightarrow (v_4)$

Figure 4: Adjacency list representation (directed graph).




$v_1 \rightarrow (v_2,\ v_3)$

$v_2 \rightarrow (v_1,\ v_3,\ v_4)$

$v_3 \rightarrow (v_1,\ v_2,\ v_6)$

$v_4 \rightarrow (v_2,\ v_5,\ v_6)$

$v_5 \rightarrow (v_4)$

$v_6 \rightarrow (v_3,\ v_6)$

Figure 5: Adjacency list representation (undirected graph).

Which one is better, adjacency matrix or adjacency list? Let's consider some measures.

- The space complexity. Clearly, the adjacency matrix needs $\Theta(|V|^2)$ space to store. The adjacency list can be stored in $\Theta(|V|+|E|)$ space, where $\Theta(|V|)$ is used to store all $|V|$ pointers, and $\Theta(|E|)$ is used to store all arrays $\{A_i\}$ as we've seen $\sum_{v_i \in V} |A_i| = |E|$. Therefore, for the space complexity, adjacency list is better, as $|E| = O(|V|^2)$; in particular in *sparse* graphs, $|E|$ will be way smaller than $|V|^2$.

- Querying if $(v_i, v_j) \in E$. Given $v_i$ and $v_j$, whether $(v_i, v_j) \in E$ can be done in $\Theta(1)$ time if the graph $G$ is represented with an adjacency matrix, as this can be answered by a direct access to $M[i, j]$. If $G$ is represented with an adjacency list, to check if $(v_i, v_j) \in E$, one needs to tranverse $A_i$ and see if $v_j$

is in $A_i$ or not, and clearly this takes $\Theta(|A_i|)$ time. Note that, if we assume that the vertices in $A_i$ are sorted, then searching for the appearance of $v_j$ in $A_i$ can be done through *binary search* which takes $\Theta(\log |A_i|)$ time. In any case, adjacency matrix is better for fast querying.

- Listing adjacent (out) vertices of a vertex. Given $v_i$, one needs to tranverse the entire row (the $i$-th row) of the adjacency matrix to find all adjacent vertices of $v_i$ which takes $\Theta(|V|)$ time. In case of adjacency list, one just needs to return the pointer to $A_i$ which takes $\Theta(1)$ time. Hence, adjacency list is better in listing adjacent vertices.

In practice, adjacency list is usually the first choice, for an obvious reason that, for huge graphs, it is not possible to store an $|V| \times |V|$ matrix in memory.

# Lecture 10 Connectivity of Undirected Graphs

A *path* (also called *walk* in some literature) in a graph $G$ is a sequence of vertices and edges, where each edge is incident to its preceding and succeeding vertices. Note that paths may contain duplicate vertices or edges. We say a path is *simple*, if it does not contain repeated vertex. If there exists a path from $u$ to $v$, then we also say $u$ can reach $v$, or $v$ is reachable from $u$, or $v$ can be reached from $u$. These definitions applies to both directed and undirected graphs. Clearly, in undirected graphs, $u$ can reach $v$ is equivalent to that $v$ can reach $u$. But this is not the case for directed graphs.

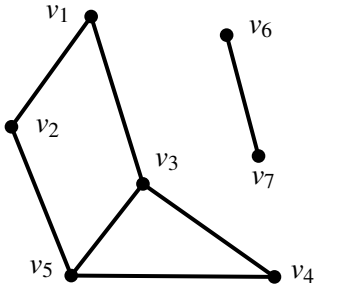
One basic procedure in graphs is to find the set of vertices that are reachable from a given vertex. We will use an array, called *visited*, of size $|V|$, to store the vertices that are reachable from the given vertex $v_i$: $visited[j] = 1$ if and only if there exists a path from $v_i$ to $v_j$. This array will be initialized as 0 for all entries. The following recursive algorithm, named *explore*, finds all vertices that are reachable from $v_i$ and stores these vertices in *visited* array properly.

```
function explore (G = (V,E), v_i ∈ V)
    visited[i] = 1;
    for each v_j where (v_i, v_j) ∈ E
        if (visited[j] = 0): explore (G, v_j);
    end for;
end algorithm;
```

In above algorithm, we can assume that $G$ is represented/stored with adjacency list. In this case, the tranverse of $v_j$ can be done by simply tranversing the list associated with $v_i$ in the adjacency list.

The time complexity of explore function is $\Theta(|E|)$, as it may traverse all lists in the adjacency list at most once, and we have showed that the total size of all lists is $\Theta(|E|)$.

Below we give two examples of running *explore* (Figure 1 and Figure 2).




input graph: $G = (V,E)$

$v_1 \rightarrow (v_2,\ v_3)$
$v_2 \rightarrow (v_1,\ v_5)$
$v_3 \rightarrow (v_1,\ v_4, v_5)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow (v_2,\ v_3,\ v_4)$
$v_6 \rightarrow (v_7)$
$v_7 \rightarrow (v_6)$

adjacency list of $G$

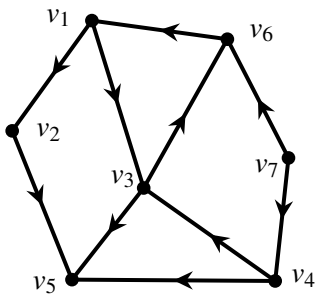
final array of *visited*

*explore* $(G,\ v_1)$

| 1 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|

Figure 1: Running *explore* $(G, v_1)$ on an undirected graph.

We now define "connected" and "connected component" to formally reveal the connectivity-structure of graphs. Let $u, v \in V$. We say $u$ and $v$ are *connected* if and only if there exists a path from $u$ to $v$ *and* there exists a path from $v$ to $u$. We note that this definition applies to both directed and undirected graph. In




input graph: $G = (V, E)$

$v_1 \rightarrow (v_2, v_3)$
$v_2 \rightarrow (v_5)$
$v_3 \rightarrow (v_5,\ v_6)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow ()$
$v_6 \rightarrow (v_1)$
$v_7 \rightarrow (v_4, v_6)$

adjacency list of $G$

*explore* $(G,\ v_1)$

final array of *visited*

| 1 | 1 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|

Figure 2: Running *explore* $(G, v_1)$ on an directed graph.

undirected graph, the existence of a path from $u$ to $v$ implies the existence of a path from $v$ to $u$. However, this is not necessarily true in directed graphs. For example, in Figure 2, there exists a path from $v_1$ to $v_5$ but there is no path from $v_5$ to $v_1$ (so they are not connected).

Let $G = (V, E)$. Let $V_1 \subset V$. We say $V_1$ is a *connected component* of $G$, if and only if (1), for *every pair of* $u, v \in V_1$, $u$ and $v$ are connected, and (2), $V_1$ is *maximal*, i.e., there does not exists vertex $w \in V \setminus V_1$ such that $V_1 \cup \{w\}$ satisfies condition (1). For example, in Figure 2, $\{v_1, v_3, v_6\}$ is a connected component; $\{v_2\}$ is a connected component; $\{v_1, v_3\}$ is not a connected component (as it is not maximal, i.e., does not satisfy condition 2).

The explore algorithm identifies all vertices reachable from a given vertex $v_i$. Hence, in the case of undirected graphs, these vertices (including $v_i$) are pairwise reachable, and these vertices are also maximal (as otherwise the explore function will find them). In other words, $explore(G, v_i)$ identifies the connected component of $G$ that includes $v_i$.

**Fact 1.** For undirected graphs, after $explore(G, v_i)$, the vertices that are marked by *visited*, i.e., $\{v_j \mid visited[j] = 1\}$ forms a connected component of $G$ that includes $v_i$.

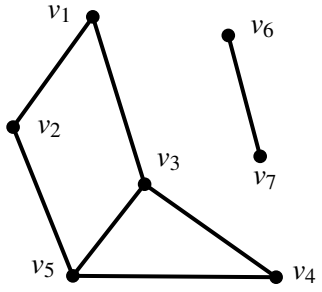
The above fact does not apply to directed graph: Figure 2 gives such an example, where $\{v_1, v_2, v_3, v_5, v_6\}$ does not form a connected component. Note: in directed graphs $\{v_j \mid visited[j] = 1\}$ are still those vertices that are reachable from $v_i$; it's just that they may not be a connected component of $G$.

How to identify *all* connected components of an undirected graph? We can run above explore algorithm multiple times, each of which starts from an un-explored vertex, until all vertices are explored. To keep track of which vertices are in which connected component, we will introduce variable *num-cc* to store the index of current connected component. We redefine the behavior of *visited* array: $visited[j] = 0$ still represents that $v_j$ has not yet been explored; $visited[j] = k$, $k \geq 1$, represents that $v_j$ has been explored and $v_j$ is in the $k$-th connected component.

This new algorithm that traverses all vertices and edges of a graph is named as DFS (depth first search). We also slightly changed the explore function, which allows to store which connected component each vertex is in. The pseudo-codes are given below.

```
function DFS (G = (V,E))
    num-cc = 0;
    visited[i] = 0, for all 1 ≤ i ≤ |V|;
    for i = 1 → |V|
        if (visited[i] = 0)
            num-cc = num-cc + 1;
            explore (G, v_i);
        end if;
    end for;
end algorithm;

function explore (G = (V,E), v_i ∈ V)
    visited[i] = num-cc;
    for each v_j where (v_i, v_j) ∈ E
        if (visited[j] = 0): explore (G, v_j);
    end for;
end algorithm;
```

Below we gave examples of running DFS on undirected graphs and undirected graphs.




input graph: $G = (V,E)$

$v_1 \rightarrow (v_2,\ v_3)$
$v_2 \rightarrow (v_1,\ v_5)$
$v_3 \rightarrow (v_1,\ v_4, v_5)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow (v_2,\ v_3,\ v_4)$
$v_6 \rightarrow (v_7)$
$v_7 \rightarrow (v_6)$

adjacency list of $G$

final array of *visited* after running $DFS(G)$

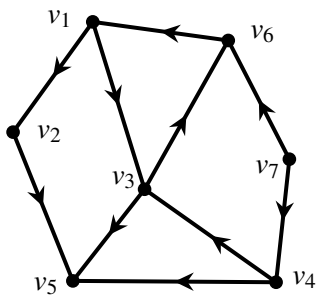
| 1 | 1 | 1 | 1 | 1 | 2 | 2 |
|---|---|---|---|---|---|---|

Figure 3: Running $DFS\ (G)$ on an undirected graph.

DFS runs in $\Theta(|E|+|V|)$ time. This is because, each vertex is explored exactly once, and each edge is examined exactly once (in the case of directed graphs) or exactly twice (in the case of undirected graphs).

**Fact 2.** For undirected graphs, DFS $(G)$ identifies all connected components of $G$: $\{v_j \mid visited[j] = k\}$ constitutes the $k$-th connected component of $G$.

Again, the above fact does not apply to directed graphs: $\{v_j \mid visited[j] = k\}$ are not necessarily form a connected component, although you can still run DFS on directed graphs. In Figure 1, $\{v_j \mid visited[j] = 1\}$ gives such an counter-example.




input graph: $G = (V, E)$

$v_1 \rightarrow (v_2, v_3)$
$v_2 \rightarrow (v_5)$
$v_3 \rightarrow (v_5,\ v_6)$
$v_4 \rightarrow (v_3,\ v_5)$
$v_5 \rightarrow ()$
$v_6 \rightarrow (v_1)$
$v_7 \rightarrow (v_4, v_6)$

adjacency list of $G$

final array of *visited* after running $DFS(G)$

| 1 | 1 | 1 | 2 | 1 | 1 | 3 |
|---|---|---|---|---|---|---|

Figure 4: Running $DFS\ (G)$ on a directed graph.

# Lecture 11 Directed Acyclic Graphs

To prepare revealing the connectivity-structure of directed graphs, we first introduce a spsecial class of directed graphs, directed acyclic graphs (DAGs).

**Definition 1** (DAG)**.** A directed graph $G = (V, E)$ is *acyclic* if and only if $G$ does not contain cycles.

Let $G = (V, E)$ be a directed graph. If a vertex $v \in V$ does not have any in-edges (i.e., *in-degree* is 0), we call it *source vertex*; if a vertex $v \in V$ does not have any out-edges (i.e., *out-degree* is 0), we call it *sink vertex*.

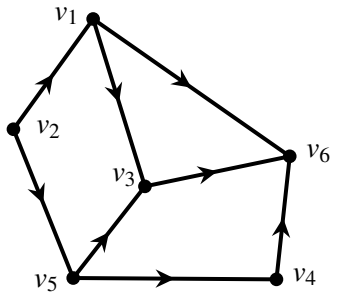
A directed graph may contain multiple source vertices or sink vertices, or may not have any source vertex or sink vertex. (Can you give such examples?)

**Claim 1.** A DAG $G = (V, E)$ always has source vertex and sink vertex.

*Proof.* Let's prove it by contradiction. Assume that $G$ does not contain any source. First, $G$ must not contain self-loop as otherwise $G$ won't be a DAG. Let $v$ be any vertex in $V$. As $v$ is not a source, we know that there exists some vertex $u$ points to $v$, i.e., $(u, v) \in E$. Now since $u$ is not a source then there must exist another vertex $w$ such that $(w, u) \in E$. Notice that $w \neq v$ as otherwise there will be a cycle: $v = w \to u \to v$. This means that $w$ is a new vertex. Again as $w$ is not a source, there must exist another *new* vertex points to it. This process can be extended infinitely following the fact and assumption that $G$ is a DAG and all vertices not are sources, but this is not possible as the number of vertices is limited. The existence of sink can be proved symmetrically. □

**Definition 2** (Linearization / Topological Sorting)**.** Let $G = (V, E)$ be a directed graph. Let $X$ be an ordering of $V$. If $X$ satisfies: if $(v_i, v_j) \in E$, then $v_i$ is before $v_j$ in $X$, then we say $X$ is a linearization (or toplogical sorting) of $G$.

See some examples below.




$(v_2, v_1, v_5, v_3, v_4, v_6)$ linearization? Yes

$(v_2, v_5, v_1, v_4, v_3, v_6)$ linearization? Yes

$(v_2, v_5, v_3, v_4, v_1, v_6)$ linearization? No, see edge $(v_1, v_3)$

$(v_2, v_1, v_4, v_5, v_3, v_6)$ linearization? No, see edge $(v_5, v_4)$

Figure 1: Examples of linearization.

If a directed graph $G$ admits a linearization, then we say $G$ can be *linearized*. We now show that linearization is an *equivalent* characterization of DAGs.

**Claim 2.** A directed graph $G$ can be linearized if and only if $G$ is a DAG.

*Proof.* Let's first prove that if $G$ can be linearized, then $G$ is a DAG. This is equivalent to proving its contraposition: if $G$ contains a cycle, then $G$ cannot be linearized. Suppose that there exists an cycle $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_k} \to v_{i_1}$ in $G$. Then the linearization $X$ must satisfy that $v_{i_j}$ is before $v_{i_{j+1}}$ for all $j = 1, 2, \cdots, k-1$, and that $v_{i_k}$ is before $v_{i_1}$, in $X$. Clearly, this is not possible.

The other side of the statement, i.e., if $G$ is a DAG, then $G$ can be linearized, can be proved constructively. We will design an algorithm (see below), that constructs a linearization for any DAG. The idea of the algorithm is to iteratively finds source vertex and removes it and its out-edges.

```
Algorithm find-linearization (G = (V, E))
    init X as empty list;
    while (G is not empty)
        arbitrarily find a source vertex u of G;
        add u to the end of X;
        update G by removing u and its out-edges;
    end while;
end algorithm;
```

This algorithm is correct. First, when a vertex $u$ is added to $X$, it is a source vertex of the current graph, which means that $\{w \mid (w,u) \in E\}$ is either empty or all of them have been added to $X$. Second, $X$ will include all vertices. This is because, a source always exists in a DAG (as we just proved). The above algorithm is more a framework, as how we update the graph is not given specifically, and which affects the running time.

Above algorithm gives a constructive proof that, a DAG can always be linearized. This completes the proof for the fact that, a directed graph is a DAG if and only if it can be linearized. □

## Meta-Graph

For a directed graph $G = (V,E)$, its structure of connectivity can be represented as a new directed graph, called *meta-graph*, denoted as $G_M = (V_M, E_M)$. Each of the vertices of the meta-graph corresponds to a connected component of $G$, and two vertices $C_i, C_j \in V_M$ are connected by edge $(C_i, C_j) \in E_M$ if and only if there exists edge $(u,v) \in E$ such that $u \in C_i$ and $v \in C_j$. An example of meta-graph is given below.

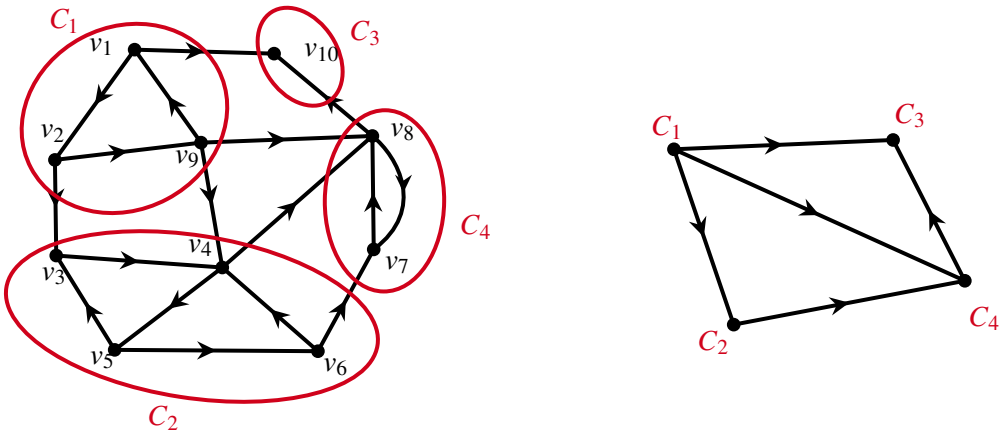



Figure 2: Example of meta-graph.

Meta-graph has an important property: it does not contain cycles, i.e., it is a directed acyclic graph (DAG).

**Claim 3.** The meta-graph $G_M$ of any directed graph $G$ is a directed acyclic graph.

*Proof.* Suppose conversely that $G_M$ contains a cycle, $C_1 \to C_2 \to C_k \to C_1$, then the union of the vertices in these connected components form a single connected component, contradicting to the *maximal* property of connected component. □

## Reverse Graph

**Definition 3.** Let $G = (V, E)$ be a directed graph. The *reverse graph* of $G$, denoted as $G_R = (V, E_R)$, has the same set of vertices and edges with reversed direction, i.e., $(u, v) \in E$ if and only if $(v, u) \in E_R$.

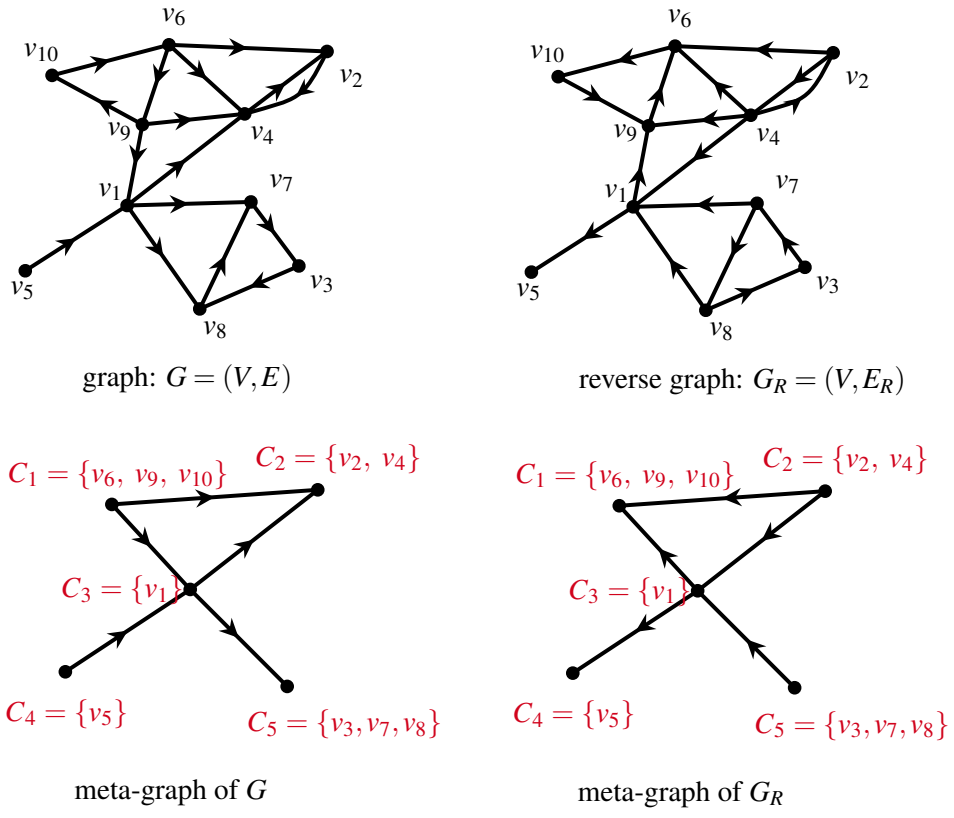



Figure 3: Graph and its reverse graph, and the corresponding meta-graphs.

Following properties can be easily proved using above definition.

**Property 1.** $(G_R)_R = G$.

**Property 2.** $X$ is a linearization of DAG $G$ if and only if the reverse of $X$ is a linearization of $G_R$.

**Property 3.** There is a path from $u$ to $v$ in $G$ if and only if there is a path from $v$ to $u$ in $G_R$. In other words, $u$ can reach $v$ in $G$ if and only if $u$ is reachable from $v$ in $G_R$.

**Property 4.** $G$ and $G_R$ has the same set of connected components.

**Property 5.** The meta-graph of $G_R$ is the reverse graph of the meta-graph of $G$. Formally, $(G_R)_M = (G_M)_R$.

# Lecture 12 Connectivity of Directed Graphs

## DFS with Timing

The DFS-with-timing is a variant of DFS that records the time of starting and finishing the explore of each vertex. It uses the following data structures (we assume $n = |V|$). These data structures are global variables, so that the explore function can get access to and edit them.

1. variable clock servers as a timer that stores the current time;
2. binary array $visited[1..n]$, where $visited[i]$ indicates if $v[i]$ has been explored/visited, $1 \leq i \leq n$;
3. array $pre[1..n]$, where $pre[i]$ records the time of starting exploring $v_i$, $1 \leq i \leq n$;
4. array $post[1..n]$, where $post[i]$ records the time of finishing exploring $v_i$, $1 \leq i \leq n$;
5. array $postlist$, stores the vertices in decreasing order of $post[\cdot]$.

The pseudo-code of DFS with timing is given below.

function DFS-with-timing $(G = (V,E))$
    $clock = 1$;
    $postlist = \emptyset$;
    $pre[i] = post[i] = -1$, for $1 \leq i \leq n$;
    for $i = 1 \to |V|$
        if $(visited[i] = 0)$: explore $(G, v_i)$;
    end for;
end algorithm;

function explore $(G = (V,E), v_i \in V)$
    $visited[i] = 1$;
    $pre[i] = clock$;
    $clock = clock + 1$;
    for any edge $(v_i, v_j) \in E$
        if $(visited[j] = 0)$: explore $(G, v_j)$;
    end for;
    $post[i] = clock$;
    $clock = clock + 1$;
    add $v_i$ to the front of $postlist$;
end algorithm;

An example of running DFS with timing is given below. Notice that this algorithm partitions all edges into two categories: solid edges $(u,v)$ implies that $v$ is visited for the first time (and therefore explore $v$ will start right now, and after exploring $v$ the program will return to explore $u$), while dashed edges $(u,v)$ implies that at that time $v$ has been visited already (and therefore $v$ will be skipped and the next out-edge of $u$ will be examined in the for-loop).

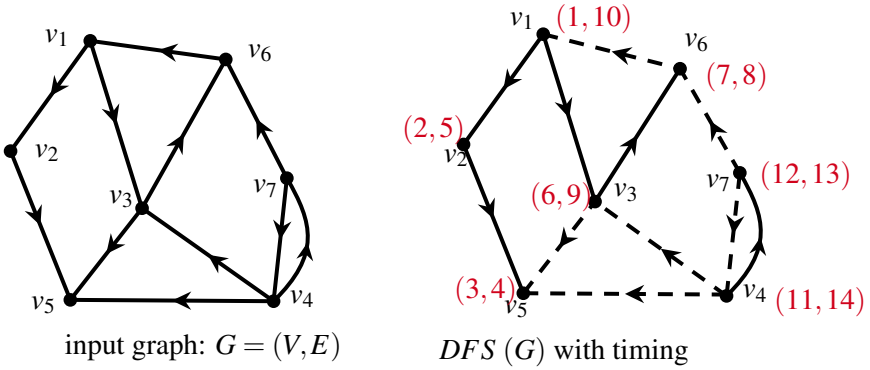



Figure 1: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $postlist = (v_4, v_7, v_1, v_3, v_6, v_2, v_5)$.

In explore $v_i$, the adjacent vertices $\{v_j \mid (v_i, v_j) \in E\}$ can be examined in any arbitrary order, i.e., all conclusions/properties we show hold regardless the order that $\{v_j\}$ gets examined. In practice though, we might follow a specific order; in Figure 1, we examine $\{v_j\}$ in increasing order of their indexes.

The above DFS-with-timing algorithm runs in $\Theta(|V| + |E|)$ time, since each vertex and each edge will be examined a constant number of times (once for directed graph, twice for undirected graph).

The above DFS-with-timing algorithm gives an interval $[pre, post]$ for each vertex. For two vertices $v_i, v_j \in V$, their corresponding intervals can either be *disjoint*, i.e., the two intervals do not overlap, or *nested*, i.e., one interval is within the other. See Figure 1. But two intervals cannot be *partially overlapping*. Why? This is because the explore funtion is recursive. There are only two possiblities that $pre[i] < pre[j]$. The first one is that explore $v_j$ is *within* explore $v_i$; in this case the recursive behaviour of explore leads to that $post[j] < post[i]$, as explore $v_j$ must terminate first and return to explore $v_i$. This case corresponds to that the two intervals are nested. The second one is that explore $v_j$ starts after explore $v_i$ finishes; this case corresponds to that the two intervals are disjoint.

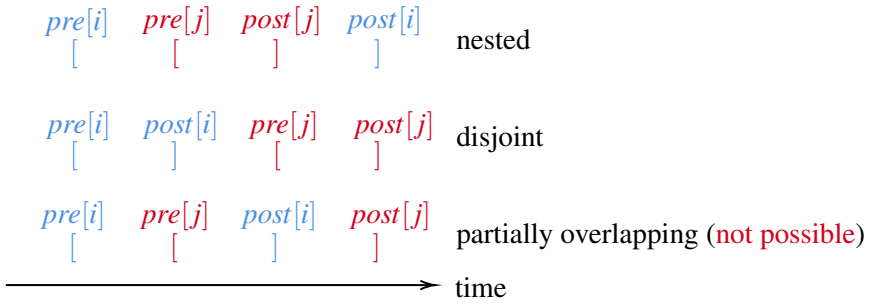



Figure 2: Relations between two $[pre, post]$ intervals.

**Claim 1.** If the $[pre[j], post[j]]$ is nested within $[pre[i], post[i]]$, then $v_j$ is reachable from $v_i$.

*Proof.* Consider when an explore will be called within another explore: only if there is an edge $(v_i, v_j) \in E$ (and $visited[j] = 0$), explore $v_j$ will be called within explore $v_i$. Consequently, the time interval for $v_j$ will be within the interval for $v_i$. Note that explore $v_j$ might call explore other vertices, such as explore $v_k$. When this happens, the time interval for $v_k$ will be within the interval for $v_j$, and therefore within the interval for $v_i$. But again this happens only if there exists edge $(v_j, v_k)$, and hence a path $v_i \to v_j \to v_k$. This argument can be extended to longer paths, proving the conclusion above. □

### Determine of Existence of Cycles

Let's see the first application of DFS-with-timing—to decide if a given (directed) graph contains cycles or not. We can simply modify the explore function, given below, and use the same DFS-with-timining function. Specifically, when the algorithm examines an edge $(v_i, v_j)$: if $v_j$ has been explored *and* its post-number has not been set yet, then the algorithm reports that $G$ contains cycle.

function explore $(G = (V,E), v_i \in V)$
  $visited[i] = 1$;
  $pre[i] = clock$;
  $clock = clock + 1$;
  for any edge $(v_i, v_j) \in E$
    if $(visited[j] = 0)$: explore $(G, v_j)$;
    else if $(post[j] = -1)$: report "$G$ contains cycle";
  end for;
  $post[i] = clock$;
  $clock = clock + 1$;
  add $v_i$ to the front of $postlist$;
end algorithm;

Now let's prove this algorithm is correct. We first prove that if $G$ contains cycle then the algorithm will always report at some time. Let $C$ be the cycle. Let $v_j \in C$ be the first vertex that is explored in $C$. Let $(v_i, v_j) \in E$ be an edge in $C$. As $v_j$ can reach $v_i$ (reason: both in cycle $C$), within exploring $v_j$ there will be a time that $v_i$ gets explored. In explore $v_i$, consider the time of examining edge $(v_i, v_j)$: at this time $visited[j]$ has been set as 1, but its post-number has not been set, as now the algorithm is still within exploring $v_j$. Therefore, the algorithm will report that $G$ contains cycle.

We then prove that if the algorithm reports, then $G$ must contain cycles. Consider that the algorithm is exploring $v_i$, examining edge $(v_i, v_j)$ and finds $visited[j] = 1$ *and* $post[j] = -1$. The fact that $post[j]$ has not been set implies that the algorithm is within exploring $v_j$. Therefore the interval for $v_i$ must be nested within the interval for $v_j$. Following Claim 1, we know that $v_j$ can reach $v_i$. In addition, there exists edge $(v_i, v_j)$. Combined, $G$ contains cycle.

Note that this algorithm is essentially determining if there exists edge $(v_i, v_j) \in E$ such that the interval $[pre[i], post[i]]$ is within interval $[pre[j], post[j]]$. (Such edges are called *back edges* in textbook [DPV], page 95.)

### Key Property of Meta-Graph

Before seeing more applications, we prove an important property that relates the post values and meta-graph.

**Claim 2.** Let $C_i$ and $C_j$ be two connected components of directed graph $G = (V,E)$, i.e., $C_i$ and $C_j$ are two vertices in its coresponding meta-graph $G_M = (V_M, E_M)$. If we have $(C_i, C_j) \in E_M$ then we must have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Intuitively, following an edge in the meta-graph, the largest post value decreases. Before seeing a formal proof, please see an example in Figure 3: the largest post values for $C_1$, $C_2$, $C_3$, and $C_4$ are 9, 6, 10, and 14, and you may verify that following any edge in the meta-graph, the largest post value always decreases.

*Proof.* Let $u^* := \arg\min_{u \in C_i \cup C_j} pre[u]$, i.e., $u^*$ is the first explored vertex in $C_i \cup C_j$. Consider the two cases.

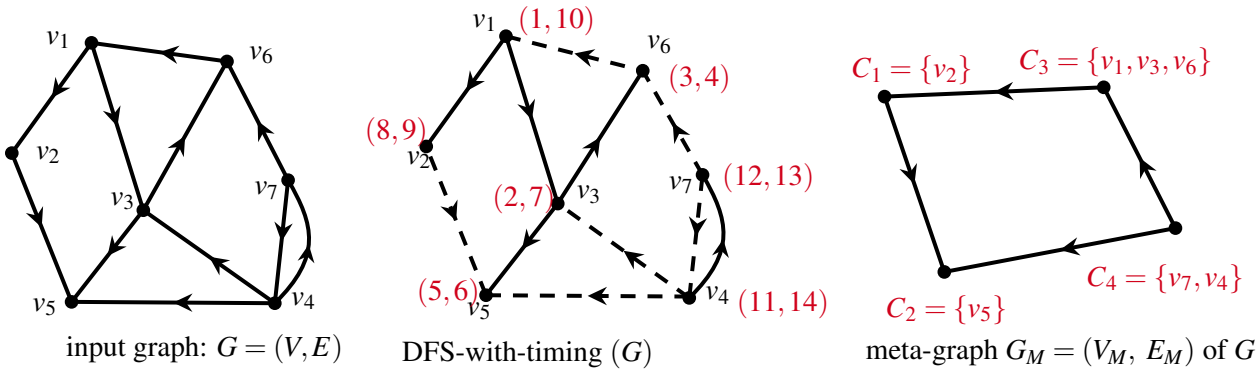



Figure 3: Example of running DFS (with timing) on a directed graph. The $[pre, post]$ interval for each vertex is marked next to each vertex.

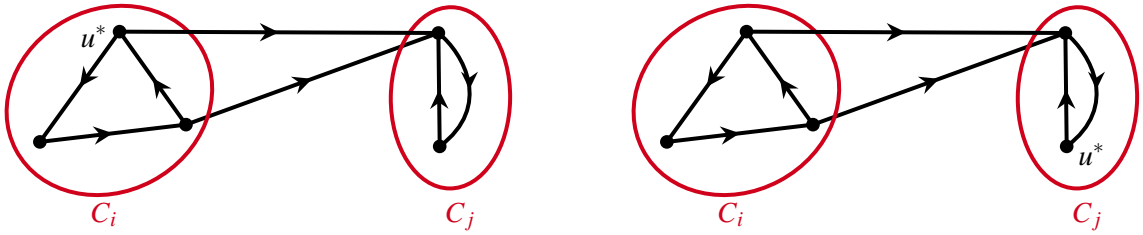



Figure 4: Two cases in proving above claim.

First, assume that $u^* \in C_i$. Then $u^*$ can reach all vertices in $C_i \cup C_j \setminus \{u^*\}$. Hence, all vertices in $C_i \cup C_j \setminus \{u^*\}$ will be explored *within* exploring $u^*$. In other words, for any vertex $v \in C_i \cup C_j \setminus \{u^*\}$, the interval $[pre[v], post[v]]$ is a subset of $[pre[u^*], post[u^*]]$. This results in two facts: $\max_{u \in C_i} post[u] = post[u^*]$ and $\max_{v \in C_j} post[v] < post[u^*]$. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$.

Second, assume that $u^* \in C_j$. Then $u^*$ can *not* reach any vertex in $C_i$; otherwise $C_i \cup C_j$ form a single connected component, conflicting to the fact that any connected component must be maximal. Hence, all vertices in $C_i$ will remain unexplored after exploring $u^*$. In other words, for any vertex $v \in C_i$, the interval $[pre[v], post[v]]$ locates after (and disjoint with) $[pre[u^*], post[u^*]]$. This gives that $\max_{u \in C_i} post[u] > post[u^*]$. Besides, we also have $\max_{v \in C_j} post[v] = post[u^*]$ as all vertices in $C_j \setminus \{u^*\}$ will be examined within exploring $u^*$. Combined, we have that $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. □

## Finding a Linearization of a DAG

The above key property holds for all directed graphs. We now consider DAGs. Note that each connected component of a DAG $G$ contains exactly one vertex, i.e., each vertex in a DAG $G$ forms the connected component of its own. (Can you spot this using Figure 5?) This is because, if a connected component contains at least two vertices $u$ and $v$ then $u$ can reach $v$ and $v$ can reach $u$ so a cycle must exist. Consequently, the meta-graph $G_M$ of any DAG $G$ is also itself, i.e., $G = G_M$.

Now let's interpret above key conclusion in the context of DAGs. For a DAG $G = (V, E)$, components $C_i$ and $C_j$ will degenerate into two vertices, say $v_i$ and $v_j$, edge $(C_i, C_j) \in E_M$ becomes $(v_i, v_j) \in E$, and $\max_{u \in C_i} post[u] = post[i]$, and $\max_{v \in C_j} post[v] = post[j]$. We have

**Corollary 1.** Let $G = (V, E)$ be a DAG. If $(v_i, v_j) \in E$, then $post[i] > post[j]$.

Now recall the definition of linearization: $X$ is a linearization if and only if for every edge $(v_i, v_j) \in E$, $v_i$ is before $v_j$ in $X$. Since it is guaranteed above that, for every edge $(v_i, v_j) \in E$, $post[i] > post[j]$, we can immediately conclude that vertices sorted in decreasing order of post-values is a linearization! This order is nothing else but the postlist.

**Corollary 2.** Let $G = (V, E)$ be a DAG. The postlist generated in the DFS-with-timing algorithm is a linearization of $G$.

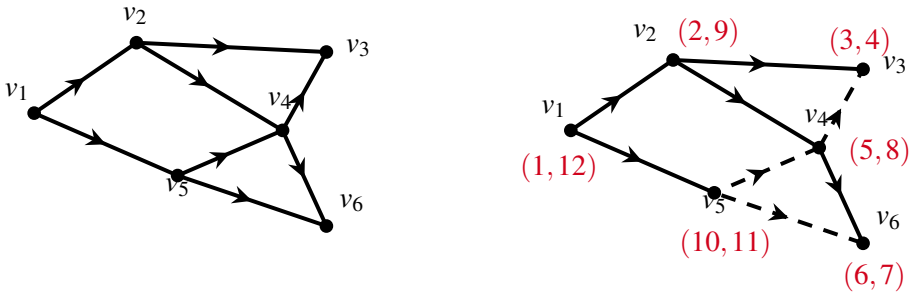


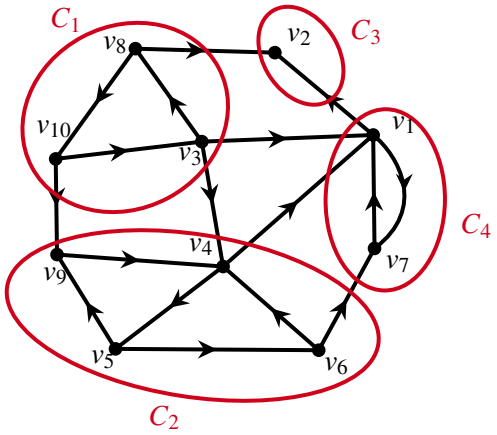

Figure 5: Example of running DFS (with timing) on a DAG $G$. The $[pre, post]$ interval for each vertex is marked next to each vertex. The *postlist* for this run is $(v_1, v_5, v_2, v_4, v_6, v_3)$, which is a linearization of $G$.

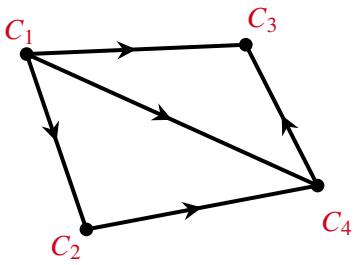
## Finding Connected Components of Directed Graphs

We now design algorithms to determine the connected components of directed graphs (i.e., the vertices of its meta-graph). Recall that the DFS algorithm introduced in Lecture 08 can successfully determine all connected components in an undirected graph. Does this algorithm also work for directed graph? And if not, where is the problem? Let's run it on an example to find a clue. See Figure 6. Recall that the (top-layer) of the DFS algorithm is to tranverse all vertices in some ordering, and if the current vertex $v$ is not yet visited, we will explore $v$. For the example in Figure 6, we run it with the (natrual) ordering $v_1, v_2, \cdots, v_{10}$.

The resulting visited array is given in the figure (please make sure you try it yourself). Unfortunately, it does not give all connected components. In fact, the vertices marked as "1"s are the union of $C_3$ and $C_4$, and the vertices marked as "2"s are the union of $C_1$ and $C_2$. The reason is quite clear: we start with explore $v_1$, and during it all vertices reachable from $v_1$ will be marked as "1" in the visited array. It turns out that $v_1$ is in connected component $C_4$, and $C_4$ can reach $C_3$ in the meta-graph. Therefore, all vertices in $C_4$ and $C_3$ are reachable from $v_1$; consequently, the visited array is set as "1" for vertices in $C_4$ and $C_3$ during explore $v_1$. Similarly, we can also see why the vertices marked as "2"s are the union of $C_1$ and $C_2$. After exploring $v_1$, $\{v_1, v_2, v_7\}$ are visited; the next unvisited vertex is $v_3$ according to above natural ordering, then the algorithm will explore $v_3$. Again, during it all (unvisited) vertices reachable from $v_3$ will be marked as "2" in the visited array. It turns out that $v_3$ is in connected component $C_1$, and $C_1$ can reach $C_2$ in the meta-graph. Therefore, all vertices in $C_1$ and $C_2$ are reachable from $v_2$; consequently, the visited array is set as "2" for vertices in $C_1$ and $C_2$.

How to fix this issue? The idea is to use a *special ordering* of vertices, instead of an arbitrary ordering, in above DFS. This special ordering should allow us to determine connected component one by one. Hence, the first vertex we explore in DFS, should be one vertex in a *sink* component of the meta-graph. In Figure 6, the only sink component is $C_3$, and since there is only one vertex in $C_3$, we should start with exploring $v_2$. Clearly, exploring $v_7$ will exactly determine the single connected component $C_2$. Next, after $C_3$ is determined, the next component we can determine is again a sink component after removing $C_3$ from the meta-graph. It is $C_4$. So, the next vertex the DFS should explore must be $v_1$ or $v_7$. It does matter which one




final array of *visited* after running DFS with ordering $(v_1, v_2, \cdots, v_{10})$:

| $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ | $v_{10}$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 2 | 2 | 1 | 2 | 2 | 2 |

final array of *visited* after running DFS with a special ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$:

| $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ | $v_{10}$ |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 4 | 3 | 3 | 3 | 2 | 4 | 3 | 4 |

Figure 6: Run the DFS algorithm with an natural (arbitrary) order and a special order (pseudo-code given below) on above example.

we pick; let's assume we pick $v_7$ (and it does not matter where $v_1$ is in the special ordering as long as it is after $v_2$ and $v_7$). Clearly, exploring $v_2$ will exactly determine the single connected component $C_4$—see the visited array. The next component we can determine is again a sink component after removing $C_3$ and $C_4$ from the meta-graph. It is $C_2$. So, the next vertex the DFS should explore must be in $\{v_4, v_5, v_6, v_9\}$. And it does matter which one we pick neither the ordering of remaining ones as long as they are behind the one we pick. Let's say we pick $v_4$. Clearly, exploring $v_4$ will exactly determine the single connected component $C_2$. Now the only component remaining is $C_1$ after removing $C_2$, $C_3$ and $C_4$ from the meta-graph. So, the next vertex the DFS should explore must be in $\{v_3, v_8, v_{10}\}$. Let's say we pick $v_8$. Clearly, exploring $v_8$ will exactly determine the last connected component $C_1$. In Figure 6, we give such a special order that follows above analysis. You should try it—run DFS following this special order correctly identifies all connected components, as shown in the Figure.

Let's summarize above observation/analysis, aiming for characterizing/constructing the special order. First, it should be clear now that, the identified connected components, following the special order, forms a reverse-linearization of the meta-graph. (Verify this with the Figure: the revealed connected components is $(C_3, C_4, C_2, C_1)$ which is a reverse-linearization of the meta-graph.) Second, the first appearance of a vertex in a connectec component matters, as exploring it identifies the entire connected component. Combined, the special ordering of vertices should satisfy this condition: *the first appearance of connected components in the special ordering should form a reverse-linearization of the meta-graph*. Again see Figure 6, the ordering $(v_2, v_7, v_4, v_6, v_8, v_1, v_3, v_9, v_{10}, v_5)$ satisfies above condition. To see that, the corresponding list of connected components is $(C_3, C_4, C_2, C_2, C_1, C_4, C_1, C_2, C_1, C_2)$, and hence the ordering of their first appearance is $(C_3, C_4, C_2, C_1)$ which is indeed a reviers-linearization of the meta-graph.

If the special ordering satisfies this condition, the DFS algorithm will work—it will identify all connected components. See the pseudo-code below. It is the same with that for undirected graphs except a single line

of difference (marked blue).

function DFS ($G = (V,E)$) *it identifies connected components for directed graphs*
    num-cc = 0;
    for $v_i$ in a specific order
        if ($visited[i] = 0$)
            num-cc = num-cc + 1;
            explore ($G, v_j$);
        end if;
    end for;
end algorithm;

function explore ($G = (V,E), v_i \in V$)
    $visited[i]$ = num-cc;
    for any edge $(v_i, v_j) \in E$
        if ($visited[j] = 0$): explore ($G, v_j$);
    end for;
end algorithm;

The remaining question is therefore how to find an ordering of vertices that satisfy above condition. Recall the key property we proved in Lecture 10: for any two connected components $C_i$ and $C_j$, if $(C_i, C_j) \in E_M$ then $\max_{u \in C_i} post[u] > \max_{v \in C_j} post[v]$. Consider the postlist, i.e., the list of vertices in descending order of post-values. The *first* appearance of (a vertex in) connected component $C_i$ in the postlist is the vertex with *largest* post-value in $C_i$, simply because postlist sorts vertices in descending order of post-values. According to the key property, if $(C_i, C_j) \in E_M$, then the first appearance of $C_i$ is before the first appearance of $C_j$. This implies the following fact.

**Claim 3.** Let $G$ be a directed graph. After running DFS-with-timing on $G$, the first appearance of connected components in the resulting postlist forms a linearization of $G_M$.

We are close. The desired special order requires "the first appearance of connected components in the special ordering form a reverse-linearization of the meta-graph", while the postlist yields "the first appearance of connected components in the postlist form a linearization of the meta-graph", How to close this last gap? The answer is the reverse-graph! We can build the reverse graph $G_R$ of the given directed graph $G$, and then run DFS-with-timing on $G_R$ to get a postlist. This postlist will be the desired, magic special order! See the pseudo-code below.

Algorithm to determine the specific order
    build $G_R$ of $G$;
    run DFS with timing on $G_R$ to get $postlist_R$;
    return $postlist_R$;
end algorithm;

We now show that, above $postlist_R$, i.e., the one obtained by running DFS-with-timing on $G_R$, satisfies the condition that "the first appearance of connected components in this $postlist_R$ forms a reverse-linearization of the meta-graph $G_M$". In fact, since this $postlist_R$ is obtained by running DFS-with-timing on $G_R$, applying Claim **??**, we know that the first appearance of connected components in $postlist_R$ forms a linearization of meta-graph of $G_R$, which is $(G_R)_M$. According to the properties of reverse graph, we know that $(G_R)_M =$

$(G_M)_R$. We also know that a linearization of $(G_M)_R$ is a reverse-linearization of $G_M$. Hence, we have that, the first appearance of connected components in $postlist_R$ forms a reverse-linearization $G_M$, proving that $postlist_R$ is the desired special order.

# Lecture 13 The 2SAT Problem

## Boolean Satisfiability Problem

The SAT problem (Boolean Satisfiability Problem) is to decide if a given boolean formula can be satisfied by assigning values to boolean variables. Let's define it. The *variables*, $x_1, x_2, \cdots, x_m$, involved are boolean (or binary), taking value of either true or false. A *literal* is either a variable itself (e.g., $x_3$, $x_5$, etc) or the negation of a variable (e.g., $\overline{x_3}$, $\overline{x_5}$, etc). A *clause* is a disjunction of literals (e.g., $x_2 \vee \overline{x_4}$, $\overline{x_3} \vee \overline{x_5} \vee x_2$, $\overline{x_1}$, etc). A *CNF (conjunctive normal form) formula* is a conjunction of clauses, such as $(x_2 \vee \overline{x_4}) \wedge (\overline{x_3} \vee \overline{x_5} \vee x_2) \wedge (\overline{x_1})$.

Given a CNF formula with $n$ clauses $C_1, C_2, \cdots, C_n$ involving $m$ variables $x_1, x_2, \cdots, x_m$, the SAT problem is to decide if there exists an *assignment* (i.e., assigning true or false value to each variable) such that all $n$ clauses are true (i.e., the CNF formula is true).

If each clause is restricted to have exactly 3 literals, then the above problem becomes the so-called 3SAT problem. 3SAT has been proved to be NP-hard, i.e., there doesn't exist any efficient algorithm for it unless P = NP. (We will introduce NP-completeness later this course; you don't need to know that these mean now.)

If each clause is restricted to have exactly 2 literals, then the above problem becomes the 2SAT problem. 2SAT can be solved efficiently, actually in $\Theta(m+n)$ time. Below we will design an algorithm for 2SAT, in which the core part is to use the algorithm of identifying connected componenets of directed graphs.

## Implication Graph and Algorithm

A clause with 2 literals represents two implication relationship. For example, consider clause $\overline{x_1} \vee x_2$. In order to satisfy it, if $\overline{x_1}$ is false then $x_2$ must be true, and if $x_2$ is false then $\overline{x_1}$ must be true. Formally, we have

**Fact 1.** Clause $A \vee B$ is equivalent to $\overline{A} \Rightarrow B$ and $\overline{B} \Rightarrow A$. Here $A$ or $B$ represents any literal (i.e., either a variable or the negation of a variable), and $\Rightarrow$ represents logic "implication".

We can then use a directed graph $G = (V, E)$, called implication graph, to represent all such implications of the given $n$ clauses (with $m$ variables). The graph contains $2m$ vertices, corresponding to all possible literals, i.e., $V = \{x_1, \overline{x_1}, x_2, \overline{x_2}, \cdots, x_n, \overline{x_n}\}$. And the graph contains $2n$ edges: each clause $A \vee B$ corresponds to 2 edges in $G$, which are $(\overline{A}, B)$ and $(\overline{B}, A)$. Intuitively, an edge $(u, v) \in E$ represents, if literal $u$ is true then literal $v$ must be true. An example of implication graph is given in Figure 1.

The implication can be carrying forward following any path in the implication graph. For example, path $u \to v \to w$ implies that if literal $u$ is true then literal $w$ must be true. What if there is path from $u$ to $\overline{u}$?

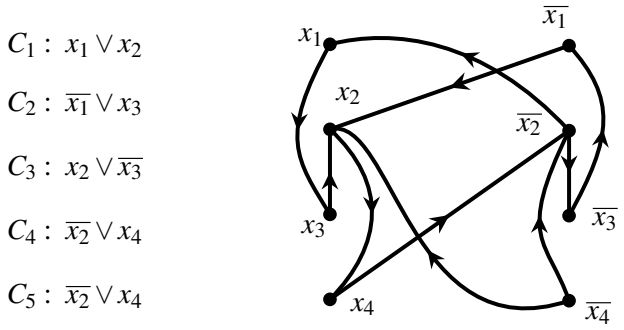



Figure 1: Graph and its reverse graph, and the corresponding meta-graphs.

This means that: if literal $u$ is true then $\overline{u}$ must be true, which is a contradiction. Hence, if such path exists then literal $u$ can't be true. What if there is path from $u$ to $\overline{u}$ *and* there is path from $\overline{u}$ to $v$? Then this means that $u$ can't be true *and* $\overline{u}$ can't be true. In other words, the instance is not satisfiable. We summarize as the following statement.

**Fact 2.** If in the implication graph there is a path from literal $u$ to literal $\overline{u}$ and there is a path from literal $\overline{u}$ to literal $u$, then the instance is not satisfiable.

The above fact gives a sufficient condition to decide if an 2SAT instance is not satisfiable. In fact, this condition is also necessary. We will prove it later on. Before that, let's design an algorithm to implement this condition, i.e., to decide if there exists a pair of literals $\{u,\overline{u}\}$ such that $u$ can reach $\overline{u}$ and $\overline{u}$ can reach $u$.

A straightforward algorithm is to explore ($G$, $u$) and explore ($G$, $\overline{u}$), for every pair of literals. This takes $O(|V|\cdot|E|)$ time, as a single run of explore takes $O(|E|)$ time. Can we do better?

Note that the condition that $u$ can reach $\overline{u}$ and $\overline{u}$ can reach $u$ is equivalent to that $u$ and $\overline{u}$ are in the same connected component of $G$. Therefore, we can use the algorithm introduced in previous lecture to determine all connected components of $G$, which takes $\Theta(|V|+|E|)$ time. Then we can simply check the resulting *visited* array: if a pair of literals satisfies $visited[u] = visited[\overline{u}]$ then the 2SAT instance is not satisfiable. The pseudo-code is given below.

```
Algorithm decide-2SAT (C_1, C_2, ..., C_n)
    build implication graph G = (V, E);
    run the DFS algorithm to find all connected components of G; this gives visited array;
    for each pair of literals {u, ū}
        if (visited[u] = visited[ū]): return false;
    end for;
    return true;
end algorithm;
```

We showed that if there exists $u$ and $\overline{u}$ that are in the same connected component, then the 2SAT instance is not satisfiable, i.e., the above algorithm is correct when returning false. We now show that if such pair does not exists, then the 2SAT instance is satisfiable, i.e., the above algorithm is correct when returning true. The proof is constructive, i.e., when such pair does not exist, we can build an assignment that satisfies all clauses. The algorithm to find such an assignment is as follows.

```
Algorithm build-assignment (C_1, C_2, ..., C_n)
    run above decide-2SAT algorithm and assume it returns true;
    build a linearization X of the meta-graph G^M of the implication graph G;
    for each pair of literals {u, ū}
        let C(u) = visited[u] and C(ū) = visited[ū] be the components that include u and ū, respectively;
        if C(u) is before C(ū) in X: set literal u to false and ū to true;
        if C(u) is after C(ū) in X: set literal u to true and ū to false;
    end for;
end algorithm;
```

Try to run above algorithm on the example given in Figure 2.

**Claim 1.** The assignment given in above algorithm satisfies all clauses.

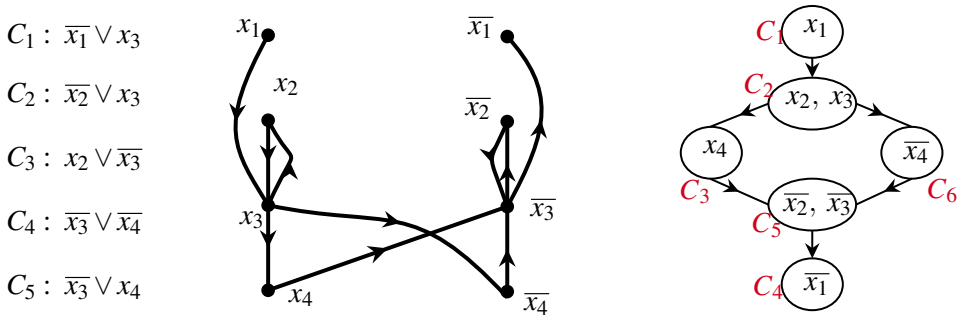



Figure 2: An instance of 2SAT, implication graph $G = (V,E)$, and the meta-graph $G^M$ of $G$. A possible linearization of $G^M$ is $X = (C_1, C_2, C_6, C_3, C_5, C_4)$. Using $X$ above algorithm gives assignment $x_1 = F$, $x_2 = F$, $x_3 = F$, and $x_4 = T$.

*Proof.* Suppose, by contradiction, that there exists a clause $u \vee v$ that is not satisfied, i.e., both literals $u$ and $v$ are set to false. According to above algorithm, we know that $C(u)$ is before $C(\overline{u})$ in $X$ and $C(v)$ is before $C(\overline{v})$ in $X$. (Note, $C(u) \neq C(\overline{u})$ and $C(v) \neq C(\overline{v})$; otherwise the decide-2SAT algorithm will return false.) Consider the construction of the implication graph $G$: this clause $u \vee v$ adds to $G$ two edges $(\overline{u}, v)$ and edge $(\overline{v}, u)$. Think about the components $C(\overline{u})$ and $C(v)$. The existence of edge $(\overline{u}, v)$ in $G$ implies that either $C(\overline{u}) = C(v)$, i.e., $\overline{u}$ and $v$ is in the same connected component, or there exists an edge from $C(\overline{u})$ to $C(v)$ in the meta-graph $G^M = (V^M, E^M)$. Hence, in any linearization $X$ of $G^M$, $C(\overline{u})$ is equal to or before $C(v)$ in $X$. Similarly, edge $(\overline{v}, u)$ in $G$ implies that either $C(\overline{v}) = C(u)$ or $(C(\overline{v}), C(u)) \in E^M$. Hence $C(\overline{v})$ is equal to or before $C(u)$ in linearization $X$.

We have four ordering constraints for $X$: $C(u)$ is before $C(\overline{u})$, $C(\overline{u})$ is equal to or before $C(v)$, $C(v)$ is before $C(\overline{v})$, and $C(\overline{v})$ is equal to or before $C(u)$. It's not possible to have $X$ to satisfy all of them. □

This claim completes the proof of the correctness of the decide-2SAT algorithm. And both decide-2SAT and build-assignment algorithms runs in $\Theta(m+n)$ time.

# Lecture 14 Queue and Priority Queue

## Queue

A *queue* data structure supports the following four operations.

1. empty ($Q$): decides if queue $Q$ is empty;
2. insert ($Q$, $x$): add element $x$ to $Q$;
3. find-earliest ($Q$): return the earliest added element in $Q$;
4. delete-earliest ($Q$): remove the earliest added element in $Q$.

To implement above operations, we can use a (dynamic) array $S$ to stores all elements, and use two pointers, *head* and *tail*, where *head* pointer always points to the first available space in $S$, and *tail* pointer always points to the earliest added element in $S$. When we add an element to $S$ we can directly add it to the place *head* points to, and when we delete the earliest added element, we can directly remove the one *tail* points to.

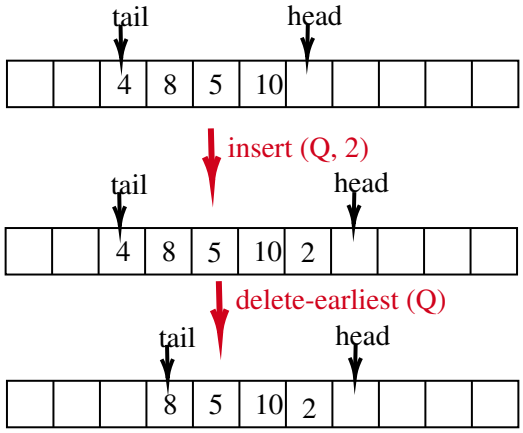



Figure 1: An example of queue.

The pseudo-code for implementing above functions are given below. Note that the input $Q$ represents above 3 data structures, the array $S$, the two pointers *head* and *tail*.

```
function empty(Q)
    if head = tail: return true;
    else: return false;
end function;

function insert(Q, x)
    S[head] = x;
    head = head + 1;
end function;

function find-earliest(Q)
    return S[tail];
end function;
```

```
function delete-earliest(Q)
    if empty(Q) = true: return;
    tail = tail + 1;
end function;
```

Note, a queue data structure exhibits a first-in-first-out property. This is because delete-earliest is the only function that removes element and it always removes the earliest added element.

## Priority Queue

In a priority queue, each element is labeled with a *priority* (also called key). The priority/key serves as the identify of the element. Additional data can be associated. A *priority queue* data structure supports the following operations.

1. empty ($PQ$): decides if priority queue $PQ$ is empty;
2. insert ($PQ$, $x$): add element $x$ to $PQ$;
3. find-min ($PQ$): return the element in $PQ$ with smallest key (i.e., highest priority);
4. delete-min ($PQ$): remove the element in $PQ$ with smallest key (i.e., highest priority).
5. decrease-key ($PQ$, pointer-to-an-element, new-key): set the key of the specified element as the given new-key.

Note that a queue can be regarded as a special case of priority queue, for which the priority is the time an element is added to the queue (hence, minimum-priority = minimum adding-time = earliest added).

There are numerous different implementations for priority queue (check wikipedia). Here we introduce one of them, *binary heap*. To do it, let's first formally introduce *heap*.

A *heap* is a (rooted) tree data structure satisfies the *heap property*. A heap is either a *min-heap* if it satisfies the *min-heap property*: for any edge $(u, v)$ in the (rooted) tree $T$, the key of $u$ (i.e., the parent) is smaller than that of $v$ (i.e., the child), or a *max-heap* if it satisfies the *max-heap property*: for any edge $(u, v)$ in the (rooted) tree $T$, the key of $u$ is larger than that of $v$. Here we always consider a heap as a min-heap, unless otherwise specified.

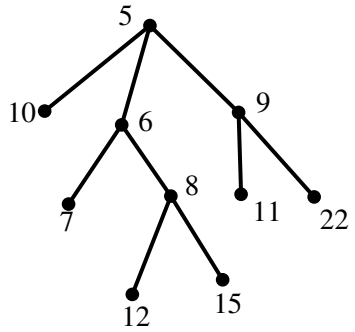



Figure 2: An example of heap. The key of an element is next to each vertex.

Because of the heap property, the root of the tree always has the smallest key. Hence, to find the element with the smallest key (i.e., find-min) one can simply return the root. This is the main reason of using a heap to implement priority queue.

A *binary heap* is a heap with the rooted tree being the *complete binary tree*. A *complete binary tree* is a binary tree (i.e., each vertex has at most 2 children) and that in every layer of the tree, except possibly the last, is completely filled, and all vertices in the last layer are placed from left to right.

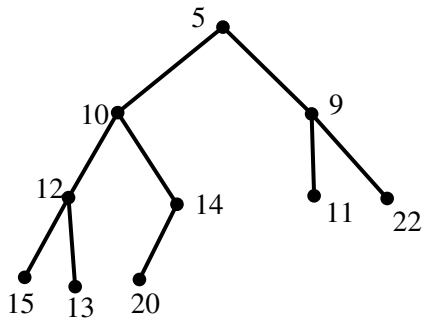



Figure 3: An example of binary heap.

Since a binary heap $T$ is so regular, we can use an array $S$ to store its elements (rather than maintaining an actual tree). The root (i.e., layer 1) of $T$ is placed in $S[1]$ (we assume that the index of $S$ starts from 1), the first element of the layer 2 is placed in $S[2]$, and so on. Generally, the $j$-th element in the $i$-th layer of $T$ will be placed in $S[2^{i-1}+j-1]$. We can also easily access the parent and children of an element:

1. the left child of $S[k]$ is $S[2k]$;
2. the right child of $S[k]$ is $S[2k+1]$;
3. the parent element of $S[k]$ is $S[\lfloor k/2 \rfloor]$.

We now introduce two common procedures used in implementing a binary heap. These procedures apply when one vertex violates the heap property, and they can adjust the heap to make it satisfy the heap property.

The *bubble-up* function applies when a vertex has a smaller key than its parent.

```
function bubble-up (S, k)
    if k ≤ 1: return;
    p = ⌊k/2⌋;
    if (S[k].key < S[p].key);
        swap S[p] and S[k];
        bubble-up (S, p);
    end if;
end function;
```

The *sift-down* function applies when a vertex has a larger key than its children. We use $n$ to denote the size of array $S$, i.e., the number of elements stored in $S$.

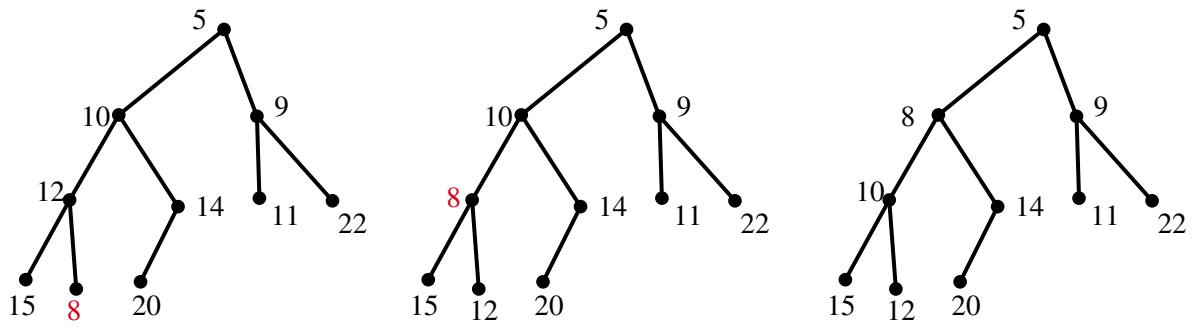



Figure 4: Illustrating bubble-up procedure.

function sift-down ($S$, $k$)
    if $2k > n$: return;
    if $2k = n$: $c = 2k$;
    else $c = \arg\min_{t \in \{2k, 2k+1\}} S[t].key$ be the index of the child of $S[k]$ with smaller key;
    if ($S[k].key > S[c].key$);
        swap $S[c]$ and $S[k]$;
        sift-down ($S$, $c$);
    end if;
end function;

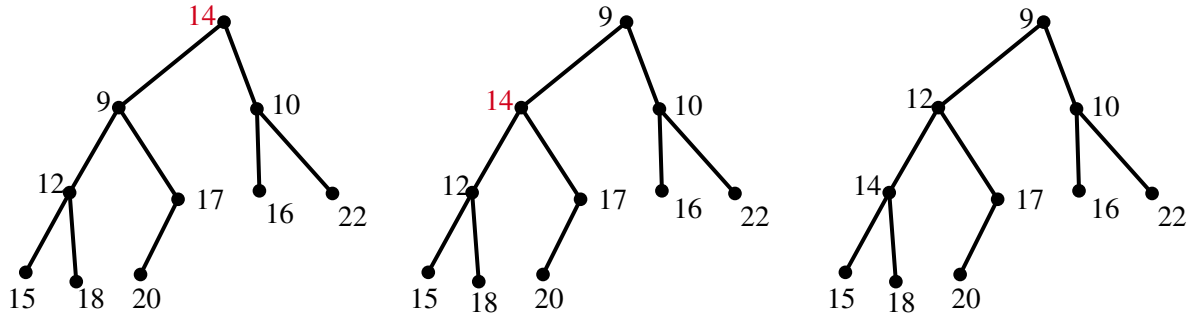



Figure 5: Illustrating sift-down procedure.

Both bubble-up and sift-down procedures runs in $O(\log n)$ time. This is because, a complete binary tree with $n$ vertices has a height of $\log n$, while the worst case of either procedure traverses along a path between the root and a leaf. Formally, in each recursive call, $k$ is either halved or doubled, and hence the number of recursive calls is $\log n$.

We finally give the pseudo-code for implementing the binary heap. In them the input $PQ$ means the array $S$ and its size $n$.

function empty($PQ$)
    if $n = 0$: return true;
    else: return true;
end function;

```
function insert(PQ, x)
    n = n + 1;
    S[n] = x;
    bubble-up (S, n);
end function;
function find-min(PQ)
    return S[1];
end function;
function delete-min(PQ)
    S[1] = S[n];
    n = n − 1;
    sift-down (S, 1);
end function;
function decrease-key(PQ, k, new-key)
    S[k].key = new-key;
    bubble-up (S, k);
end function;
```

The empty and find-min procedures takes $\Theta(1)$ time; the other 3 procedures takes $O(\log n)$ time, as they are dominated by either bubble-up or sift-down.

# Lecture 15 Shortest Path Problems and BFS

In many applications, edges of graphs are associated with *lengths*. We will consider three cases.

1. unit edge length: the length of each edge is 1;
2. positive edge length: the length of each edge is a positive number;
3. length of edge might be a negative number.

Let $G = (V,E)$ be a graph; let $l(e)$ be the length of $e \in E$ (in any of above cases). Let $p$ be a path of $G$. We define the length of path $p$, still denoted as $l(p)$, as the sum of the length of all edges in $p$, i.e., $l(p) := \sum_{e \in p} l(e)$. Given $u, v \in V$, the *shortest path* from $u$ to $v$, is the path from $u$ to $v$ with smallest length. We use notation $distance(u,v)$ to denote the length of the shortest path from $u$ to $v$.

There are different variants of shortest path problems. One category is *single-source shortest path problem*, where we are given graph $G = (V,E)$ with edge length (one of the three cases) and a *source* vertex $s \in V$, and we seek to find the shortest path from $s$ to *all* vertices.

We first solve the easiest version of shortest path problem: given $G = (V,E)$ with unit edge length and a *source* vertex $s \in V$, to find the shortest path from $s$ to all vertices, i.e., to find $distance(s,v)$ for any $v \in V$.

The breadth-first search (BFS) can solve above problem. The idea of BFS is to traverse the vertices of graph in increasing order of their distance from $s$. Formally, we define $V_k$, $k = 0,1,2,\cdots$, as the subset of vertices whose distance (from $s$) is exactly $k$, i.e., $V_k = \{v \in V \mid distance(s,v) = k\}$. BFS will first traverse vertices in $V_0$, then vertices in $V_1$, then $V_2$, and so on, until all vertices reachable from $s$ are traversed.

Notice that $V_0 = \{s\}$ as $distance(s,s) = 0$. How about $V_1$? They are the vertices reachable from $s$ with one edge (but cannot be $s$), i.e., $V_1 = \{v \in V \mid (s,v) \in E\} \setminus V_0$. How about $V_2$? They are the vertices reachable from some vertex in $V_1$ with one edge, but not in $V_0$ or $V_1$, i.e., $V_2 = \{v \in V \mid u \in V_1 \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1)$. The general form is that $V_{k+1} = \{v \in V \mid u \in V_k \text{ and } (u,v) \in E\} \setminus (V_0 \cup V_1 \cup \cdots \cup V_k)$. This suggests an iterative framework of BFS: to find $V_{k+1}$, explore the out-edges of $V_k$ to collect the *newly* reached vertices (i.e., those not in $V_0 \cup V_1 \cup \cdots \cup V_k$). To realize this idea, BFS uses two data structures; the complete algorithm follows.

1. Array $dist$ of size $|V|$, initialized as $dist[v] = \infty$ for any $v \in V$. Array $dist$ serves as two purposes: indicating a vertex has been reached or not, and storing the distance from $s$ (after it has been reached). Formally, before $v$ is reached, $dist[v] = \infty$; after $v$ is reached, $dist[v] = distance\ (s,v)$.
2. Queue $Q$, which stores the vertices haven't been explored. Vertices will be added to $Q$ in the order of $V_0, V_1, V_2, \cdots$, and vertices will be deleted from $Q$ in the same order (as queue is first-in-first-out).

Initially BFS has $V_0$ (i.e., $s$) in $Q$ (right before the while loop), then BFS deletes $s$ from $Q$ and explores $s$ (newly reached vertices—these are $V_1$, will have their $dist$ set as 1 and be added to $Q$); at the time of finishing exploring $V_0$, $V_1$ will be in $Q$. Next, BFS will gradually delete and explore each of the vertices in $Q$ (i.e., $V_1$); in this process, vertices in $V_2$ will be reached, their $dist$ be set as 2, and be added to $Q$; after all of them are deleted and explored, $Q$ will exactly consist of $V_2$.

In general, BFS keeps the following invariant: for every $k = 0,1,2,\cdots$, there is a time at which $Q$ contains exactly $V_k$, $dist[v] = distance(s,v)$ for any $v \in V_0 \cup V_1 \cup \cdots V_k$, and $dist[v] = \infty$ for all other vertices. This invariant explains the behavior of BFS while also proves its correctness: when the $Q$ becomes empty, $dist[v] = distance(s,v)$ for any $v \in V$.

Following above invariant and the pseudo-code, we know that the time that vertex $v$ is reached for the first

time (happend when $(u,v) \in E$ is checked and $dist[v] = \infty$), is also the time that the shortest path to $v$ is found, that the $dist[v]$ gets assigned, and that $v$ is added to $Q$.

```
Algorithm BFS (G = (V,E), s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    init an empty queue Q;
    dist[s] = 0;
    insert (Q, s);
    while (empty (Q) = false)
        u = find-earliest (Q);
        delete-earliest (Q);
        for each edge (u, v) ∈ E
            if (dist[v] = ∞)
                dist[v] = dist[u] + 1;
                insert (Q, v);
            end if;
        end for;
    end while;
end algorithm;
```

BFS runs in $O(|V|+|E|)$ time, as each vertex will be explored at most once, and each edge will be examined at most once (for directed graph) and at most twice (for undirected graph). Note that BFS ($G$, $s$) only traverses those vertices in $G$ can be reached from $s$.

# Lecture 16 Dijkstra's Algorithm

We now study the single-source shortest path problem with positive edge length: given graph $G = (V, E)$ with edge length $l(e) > 0$ for any $e \in E$ and source vertex $s \in V$, to seek $distance(s, v)$ for any $v \in V$. We solve this problem with the *Dijkstra's algorithm*.

Similar to BFS, the idea of Dijkstra's algorithm is also to determine and calculate the distance of each vertex in increasing order of distance. More specifically, let $R = (v_1^*, v_2^*, \cdots, v_n^*)$, $n = |V|$, be the order of vertices with increasing distance, i.e., $distance(s, v_k^*) \leq distance(s, v_{k+1}^*)$, $1 \leq k < n$. In other words, we define $v_k^*$ as the $k$-th closest vertex from $s$. We don't know this order in advance. But Dijktra's algorithm will identify vertices in this order and calculate their distance. For the sake of writing and notations, we define $R_k = \{v_1^*, v_2^*, \cdots, v_k^*\}$, i.e., the first $k$ vertices in above list; $R_k$ are also the closest $k$ vertices from $s$.

Clearly, $v_1^* = s$, $distance(s, v_1^*) = distance(s, s) = 0$, and $R_1 = \{s\}$. The key question is: given $R_k$ (i.e., the closest $k$ vertices from $s$) and their distances (i.e., $distance(s, v)$ for every $v \in R_k$), how to find $v_{k+1}^*$? Below we show that $v_{k+1}^*$ must be within *one-edge extension* of $R_k$.

**Claim 1.** There must exist vertex $u \in R_k$ such that $(u, v_{k+1}^*) \in E$.

*Proof.* Suppose conversely that for every $u \in R_k$ we have $(u, v_{k+1}^k) \notin E$. Consider the shortest path from $s$ to $v_{k+1}^*$. Let $w$ be the vertex right before $v_{k+1}^*$ in path $p$, i.e, $(w, v_{k+1}^*)$ is the last edge of $p$. We have $distance(s, v_{k+1}^*) = distance(s, w) + l(w, v_{k+1}^*)$. As edges have positive edge length, we have $l(w, v_{k+1}^*) > 0$, and consequently $distance(s, w) < distance(s, v_{k+1}^*)$. Besides, according to the assumption, we have $w \notin R_k$. Therefore, $v_{k+1}^*$ cannot be the $(k+1)$-th closest vertex from $s$, because $w$ has a shorter distance than $v_{k+1}^*$. This contradicts to the definition of $v_{k+1}^*$. □

Note that, in above proof, we use the fact that edges have positive lengths. Hence, above claim may not be true for graphs with negative edge length. This also explains why Dijkstra's algorithm does not work for graphs with negative edge length.

The above claim shows that, the last edge $(u, v)$ of the shortest path from $s$ to $v_{k+1}^*$ satisfies that $u \in R_k$ and $v \notin R_k$. Which such edge is the correct one? We don't know, so we enumerate all such edges $(u, v) \in E$ with $u \in R_k$ and $v \notin R_k$. Suppose that $(u, v)$ is the last edge of the shortest path from $s$ to $v_{k+1}^*$, we know that $distance(s, v) = distance(s, u) + l(u, v)$. This leads to the following formula to calculate $v_{k+1}^*$ and its predecessor in the shortest path. See an example in Figure 1.

**Corollary 1.** We have

$$distance(s, v_{k+1}^*) = \min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s, u) + l(u, v)).$$

Let $(u', v')$ be the optimal edge in the minimization, i.e.,

$$(u', v') := \arg\min_{u \in R_k, v \in V \setminus R_k, (u,v) \in E} (distance(s, u) + l(u, v)).$$

Then $v_{k+1}^* = v'$ and $u'$ is the predecessor of $v_{k+1}^*$ in the shortest path from $s$ to $v_{k+1}^*$.

We therefore have the following algorithm (framework), in which we follow above formula to iteratively construct $R_k$, $k = 1, 2, \cdots, n$. We again use array *dist* of size $n$ to store the distance for vertices in $R_k$.

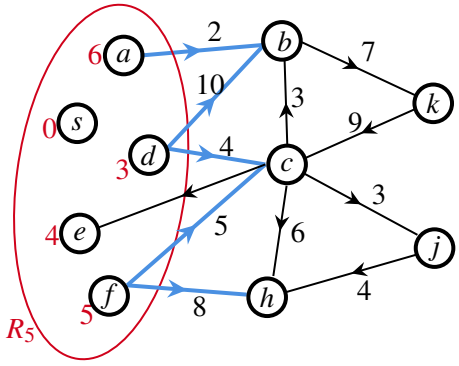



Figure 1: Example for Colollary 1. Suppose that we know $R_5 = \{s,d,e,f,a\}$ and their distances from $s$, marked next to vertices. To find $v_6^*$ and calculate $distance(s,v_6^*)$, we consider all one-edge extension of $R_5$, marked as thick blue edges. Following Corollary 1, $distance(s,v_6^*) = \min_{u\in R_5, v\notin R_5,(u,v)\in E}(distance(s,u) + l(u,v)) = \min\{6+2, 3+10, 3+4, 5+5, 5+8\} = 7$, and the optimal edge is $(d,c)$. Hence, $v_6^* = c$ and $d$ is the predecessor of $d$ in the shortest path.

Algorithm Dijkstra-Framework $(G = (V,E), s \in V)$
  let $R_1 = \{s\}$;
  $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
  for $k = 1 \to n-1$
    calculate $(u',v') := \arg\min_{u\in R_k, v\in V\setminus R_k,(u,v)\in E}(dist[u] + l(u,v))$;
    $R_{k+1} = R_k \cup \{v'\}$;
    $dist[v'] = dist[u'] + l(u',v')$;
  end for;
end algorithm;

A naive implementation of above framework takes $O(|V|\cdot|E|)$ time, as for each $k$, the calculation of the minimization may take $O(|E|)$ time. Below we show how to use efficient data structures to speed up, leading to the complete Dijkstra's algorithm with improved running time.

We rewrite the formula in Corollary 1 into an equivalent form by separating the minimization into two levels:

$$distance(s,v_{k+1}^*) = \min_{v\in V\setminus R_k} \min_{u\in R_k,(u,v)\in E}(distance(s,u) + l(u,v)).$$

Now we define the inner level of minimization with a new term $dist_k(v)$:

$$dist_k(v) := \min_{u\in R_k,(u,v)\in E}(distance(s,u) + l(u,v)).$$

Then we have

$$distance(s,v_{k+1}^*) = \min_{v\in V\setminus R_k} dist_k(v).$$

Intuitively, $dist_k(v)$ gives the length of the shortest path from $s$ to $v$ by only using vertices in $R_k$. This means that $dist_k(v)$ is always an upper bound of the distance, formally $dist_k(v) \geq distance(s,v)$. This is because, $dist_k(v)$ represents the length of the optimal path of a subset of all possible paths from $s$ to $v$, hence giving an upper bound.

With $dist_k$ available, the next closest vertex, i.e., $v_{k+1}^*$ can be easily calculated by picking the one with smallest $dist_k$ value. And the distance is equal to the corresponding dist value: $distance(s,v_{k+1}^*) = dist_k(v_{k+1}^*)$.

Try above formulas in Figure 1. Answer: $dist_5(b) = \min\{6+2, 3+10\} = 8$, $dist_5(c) = \min\{3+4, 5+5\} = 7$, $dist_5(h) = 5+8 = 13$, $dist_5(k) = \infty$, $dist_5(j) = \infty$; hence $v^*_6 = c$ and $distance(s, v^*_6) = dist_5(c) = 7$.

The reason we introduce $dist_k$ is that, $dist_{k+1}$ can be calculated in an incremental way by largely reusing $dist_k$. This is much more efficiently than directly calculating $dist_{k+1}$ using the definition. To see the details, recall its definition, for any $v \in V \setminus R_{k+1}$, we have

$$dist_{k+1}(v) = \min_{u \in R_{k+1}, (u,v) \in E}(distance(s,u) + l(u,v)).$$

Note that $R_{k+1} = R_k \cup \{v^*_{k+1}\}$. Hence

$$dist_{k+1}(v) = \begin{cases} \min\{dist_k(v), distance(s, v^*_{k+1}) + l(v^*_{k+1}, v)\} & \text{if } (v^*_{k+1}, v) \in E \\ dist_k(v) & \text{if } (v^*_{k+1}, v) \notin E \end{cases}$$

In other words, when calculating $dist_{k+1}$, we only need to examine the out-edges of $v^*_{k+1}$ and update only if the use of $v^*_{k+1}$ leads to a shorter path. The pseudo-code of calculating $dist_{k+1}$ from $dist_k$ is given below. Before calling this updating procedure, we assume $dist_{k+1}$ is initialized as $dist_k$; that is the update-dist procedure below just update for vertices whose dist values might be reduced. (In the complete algorithm, you will see that we will use one *dist* array for all $k$, hence such initialization is not necessary.)

```
procedure update-dist (v*_{k+1})
    for (v*_{k+1}, v) ∈ E
        if (dist_k(v*_{k+1}) + l(v*_{k+1}, v) < dist_{k+1}(v))
            dist_{k+1}(v) = dist_k(v*_{k+1}) + l(v*_{k+1}, v);
        end if;
    end for;
end procedure;
```

Try above procedure with the example below.

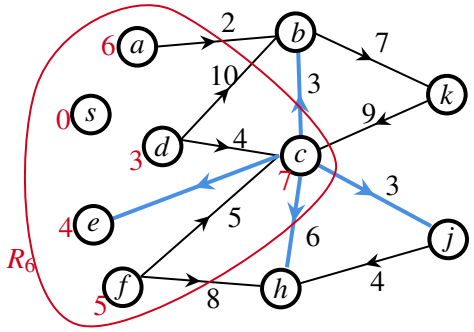



Figure 2: Following Figure **??**, we have that $v^*_6 = c$. We now want to calculate $dist_6$ using $dist_5$. We consider the out-edges of $c$, marked as thick blue edges. We have, $dist_6(b) = \min\{dist_5(b), 7+3\} = \min\{8, 10\} = 8$, $dist_6(h) = \min\{dist_5(h), 7+6\} = \min\{13, 13\} = 13$, $dist_6(k) = dist_5(k) = \infty$, $dist_6(j) = \min\{dist_5(j), 7+3\} = \min\{\infty, 10\} = 10$.

The last piece of Dijkstra's algorithm comes with the use of *priority queue* to quickly pick the next closest vertex, i.e., to calculate $v^*_{k+1} = \arg\min_{v \in V \setminus R_k} dist_k(v)$. To this end, the priority queue $PQ$ always stores $V \setminus R_k$, and for each vertex $v$ that is stored in $PQ$, its priority is $dist_k(v)$. In this way, every time we call find-min ($PQ$), it gives us $\min_{v \in V \setminus R_k} dist_k(v)$.

The pseudo-code for complete Dijkstra's algorithm is given below. First, instead of maintaining a $dist_k$ array for each separate $k$, we just need to maintain a single *dist* array (the index $k$ will increase implicitly when the next closest vertex gets identified and removed from the priority-queue). Second, we do not need to implicitly maintain $R_k$, as $PQ$ is always complement to $R_k$. In order to maintain this invariant, we delete the next closed vertex $u$, by calling delete-min, at the time of identifying $u$. In the update-list procedure, in order to guarantee that the priority of $v$ is always $dist(v)$, we call decrease-key every time we update $dist(v)$.

Recall that Dijkstra's algorithm sequentially identifies the closests vertices from $s$. Formally, if $u$ is removed from $PQ$ before $v$, then $distance(s,u) \leq distance(s,v)$.

Where are the final distances from $s$? They are in array *dist*. This is because, at the time a vertex $u$ is picked by find-min and removed from $PQ$, *dist* value for this vertex $u$ is exactly its distance, i.e., $dist(u) = distance(s,u)$. This dist value for $u$ will remain unchanged till the end of the algorithm. Notice though, later in the algorithm, when $v$ is picked by find-min (and removed from $PQ$), there might be an edge $(v,u)$ and therefore $dist(v)+l(v,u)$ will be compared with $dist(u)$ according to the algorithm, and if the former is smaller than the later, $dist(u)$ will be reduced/changed. (See an example in Figure 2, edge $(c,e)$.) But such change will never happen. This is because, $u$ is picked before $v$, implying that $distance(s,u) \leq distance(s,v)$. Hence $dist(v)+l(v,u) = distance(s,v)+l(v,u) > distance(s,v) \geq distance(s,u) = dist(u)$. So, the update to $dist(u)$ will not happen.

Another way to understand why such update to $dist(u)$ will not happen is that $dist(u)$ is always an upper bound of $distance(s,u)$. When $u$ gets removed from $PQ$, this bound is reached, i.e., $dist(u) = distance(s,u)$. Hence after that $dist(u)$ will remain this minimized value and hence cannot get further reduced.

```
Algorithm Dijkstra (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    init an empty priority queue PQ;
    for any v ∈ V: insert (PQ, v), where the priority of v is ∞;
    dist[s] = 0;
    decrease-key (PQ, s, 0);
    while (empty (PQ) = false)
        u = find-min (PQ);
        delete-min (PQ);
        for each edge (u,v) ∈ E
            if (dist[v] > dist[u] + l(u,v))
                dist[v] = dist[u] + l(u,v);
                decrease-key (PQ, v, dist[v]);
            end if;
        end for;
    end while;
end algorithm;
```

The running time of Dijkstra's algorithmd depends on the specific implementation of priority queue used. Consider using binary heap. The break-down of running time is given below. Note that each vertex will be picked from the $PQ$ at most once and each edge will be examined at most once (for directed graph) or at most twice (for undirected graph). The total running time is $O((|V|+|E|)\log|V|)$.

1. initialization: $\Theta(|V|)$;

2. insert ($PQ$): $|V| \times O(\log |V|)$;
3. empty ($PQ$): $|V| \times \Theta(1)$;
4. find-min ($PQ$): $|V| \times \Theta(1)$;
5. delete-min ($PQ$): $|V| \times O(\log |V|)$;
6. updating-dist: $|E| \times \Theta(1)$;
7. decrease-key ($PQ$): $|E| \times O(\log |V|)$;

# Lecture 17 Bellman-Ford Algorithm

Bellman-Ford algorithm can be used to solve the (single-source) shortest path problem with negative edge length, and its extension can also be used to detect if a graph contains negative cycle (reachable from the given source).

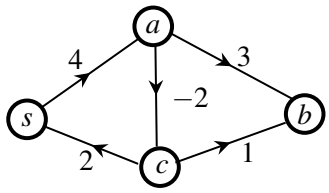
Bellman-Ford algorithm is quite simple. It only maintain an array, *dist* of size $|V|$, as its data structure. And it just does a bunch of "update" operations. An "update" function takes an edge $e = (u,v)$ as input, and updates $dist[v]$ as $dist[u] + l(u,v)$ if the former is larger than the latter.

```
procedure update(edge (u,v) ∈ E)
    if (dist[v] > dist[u] + l(u,v))
        dist[v] = dist[u] + l(u,v);
    end if;
end procedure;
```

Bellman-Ford algorithm iterates $(|V|-1)$ rounds, and in each round, updates *all* edges, in an arbitrary order.

```
Algorithm Bellman-Ford (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init an array dist of size |V|;
    dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    for k = 1 → |V| − 1
        for each edge (u,v) ∈ E
            update(u,v);
        end for;
    end for;
end algorithm;
```

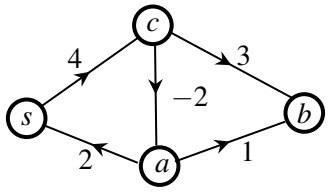
See two examples of running Bellman-Ford algorithm below.




graph

| | s | a | b | c |
|---|---|---|---|---|
| *init* | 0 | ∞ | ∞ | ∞ |
| *R*1 | 0 | 4 | ∞ | ∞ |
| *R*2 | 0 | 4 | 3 | 2 |
| *R*3 | 0 | 4 | 3 | 2 |

dist array




graph

| | s | a | b | c |
|---|---|---|---|---|
| *init* | 0 | ∞ | ∞ | ∞ |
| *R*1 | 0 | ∞ | ∞ | 4 |
| *R*2 | 0 | 2 | 7 | 4 |
| *R*3 | 0 | 2 | 3 | 4 |

dist array

Figure 1: The dist array (after each round) running Bellman-Ford algorithm on each example. In each round, we choose to update all edges in lexicographic order, i.e., $(a,b),(a,c),(c,b),(c,s),(s,a)$.

Since update function takes constant time, clearly, Bellman-Ford algorithm runs in $\Theta(|V| \cdot |E|)$ time.

To prove the correctness of Bellman-Ford algorithm, we first introduce some definitions and prove some properties.

A negative cycle $C$ in a graph is a cycle with negative length, i.e., $l(C) := \sum_{e \in C} l(e) < 0$. In the presence of negative cycle, if we do not limit the number of edges in a path, then the length of a path could goes to

negative infinity. In other words, the shortest path may not exist in graphs with negative cycles. We therefore need to be careful when talking about shortest paths or $distance(\cdot,\cdot)$, as they have to exist first. It suffices to require the graph to be negative cycle free. Later in this Lecture we will also introduce an algorithm to detect if a given graph contains negative cycle or not.

A path $p$ in a graph is *simple* if $p$ does not have repeating vertices. If a graph $G$ does not contain negative cycle, then for any pair of vertices $u$ and $v$, if $u$ can reach $v$, then there always exists a simple shortest path from $u$ to $v$, as otherwise we can skip the cycle in it to get a better or same-length path. If all cycles in graph $G$ are positive then every shortest path is simple. Since any simple path contains at most $|V|-1$ edges, we have the following:

**Fact 1.** If $G$ does not contain negative cycles, for every $u,v \in V$ where $v$ is reachable from $u$, there exists a shortest path from $u$ to $v$ with at most $(|V|-1)$ edges.

Shortest path admits the following *optimal substructure* property. Intuitively, this property states that, the shortest path from $u$ to $v$ contains the shortest path from $u$ to any internal vertex on this path (formally described below). Essentially, this is why shortest path problem can be solved efficiently.

**Fact 2.** Let $p = v_1 \to v_2 \to v_3 \to \cdots \to v_{k-1} \to v_k$ be the shortest path from $v_1$ to $v_k$. Then for any $1 \le i \le k$, $p_i = v_1 \to v_2 \to \cdots \to v_i$, i.e., the portion of $p$ from $v_1$ to $v_i$, is the shortest path from $v_1$ to $v_i$.

*Proof.* Suppose that $p_i = v_1 \to v_2 \to \cdots \to v_i$ is not the shortest path from $v_1$ to $v_i$. Assume that $q$ is the shorest path from $v_1$ to $v_i$. Then we can construct a path from $v_1$ to $v_k$ shorter than $p$, by concatenating $q$ and $v_i \to v_{i+1} \to \cdots \to v_k$. This contradicts to the fact that $p$ is the shortest path from $v_1$ to $v_k$. □

The above fact immediately implies the following:

**Fact 3.** If there exists one shortest path from $s$ to $v$ such that $(u,v)$ is the last edge on this shortest path, then we have that $distanct(s,v) = distance(s,u) + l(u,v)$.

Now let's go back to the Bellman-Ford algorithm. We first show an invariant about its data structure the *dist* array:

**Fact 4.** Throughout the Bellman-Ford algorithm, if $dist[v] \neq \infty$ then $dist[v]$ represents the length of some path from $s$ to $v$.

*Proof.* Clearly, in the initialization step which sets $dist[s] = 0$ and $dist[v] = \infty$ for all $v \neq s$, above claim holds, as $dist[s]$ stores a path from $s$ to $s$ without any edge and therefore its length is 0. Now to show above fact is correct throughout the algorithm, we just need to show that the "update" operation keeps this invariant (as this algorithm does nothing else but "update"). We can prove this by induction. Assume that up to the $n$-th update opertion above claim holds, i.e., $dist[v]$ stores the length of some path from $s$ to $v$ when $dist[v] \neq \infty$. Now consider the $(n+1)$-th update operation on edge $e = (u,v)$. Assume that $dist[v] > dist[u] + l(u,v)$, as otherwise this operation does not change *dist* and the claim continues to be true. Now $dist[v]$ is updated as $dist[u] + l(u,v)$. Since, according to the inductive assumption, $dist[u]$ stores the length of some path from $s$ to $u$, we have that $dist[v]$ stores the length of the path that consists of the aforementioned path from $s$ to $u$ followed by edge $(u,v)$. □

Assume that the shortest path from $s$ to $v$ exists. Following above fact, $dist[v] \geq distance(s,v)$ throughout the algorithm, as $dist[v]$ represents the length of *some* path from $s$ to $v$, while $distance(s,v)$ represents the length of the *shortest* path from $s$ to $v$. In Bellman-Ford algorithm, $dist[v]$ starts from a trivial upper bound (i.e., infinity) of $distance(s,v)$, and will get closer and closer to $distance(s,v)$ through updates, and eventually reach $distance(s,v)$. We now state the conditions for this to happen.

**Fact 5.** If edge $(u,v)$ is the last edge on one shortest path from $s$ to $v$ and $dist[u] = distance(s,u)$, then after

$update(u,v)$ we will have $dist[v] = distance(s,v)$.

*Proof.* Since edge $(u,v)$ is the last edge on one shortest path from $s$ to $v$, according to Fact 3, we know that $distance(s,v) = distance(s,u) + l(u,v) = dist[u] + l(u,v)$. Through $update(u,v)$, $dist[v]$ will be compared with $dist[u] + l(u,v) = distance(s,v)$. The first case will be that $dist[v] \leq dist[u] + l(u,v) = distance(s,v)$. Notice that in this case we must have $dist[v] = distance(s,v)$ according to Fact 1, i.e., $dist[v]$ already stores the distance. The second case will be that $dist[v] > dist[u] + l(u,v) = distance(s,v)$, and in this case the "update" function will set $dist[v] = dist[u] + l(u,v) = distance(s,v)$. Hence, in either case, we will have $dist[v] = distance(s,v)$ after updating edge $(u,v)$. □

Suppose that $s \to v_1 \to v_2 \to \cdots \to v$ is one shortest path from $s$ to $v$. In the initialization step we have $dist[s] = distance(s,s) = 0$. If at a later time, $update(s,v_1)$ is executed, then following above Fact 5, we know that $dist[v_1] = distance(s,v_1)$ after this update (reasoning: $dist[s] = distance(s,s)$, and $(s,v_1)$ is the last edge on one shortest path from $s$ to $v_1$ according to the optimal substrcture property). Once $dist[v_1]$ becomes $distance(s,v_1)$, $dist[v_1]$ will stay as $distance(s,v_1)$ according to Fact 1. If at a later time $update(v_1,v_2)$ happens then following Fact 5, we have that $dist[v_2] = distance(s,v_2)$. Note that it does not matter if additional updates happen between $update(s,v_1)$ and $update(v_1,v_2)$. We can continue this argument; a general form is summarized below.

**Fact 6.** If there exists a sequence of udpate procedures (not necessarily consecutive, i.e., there can be update(s) between any two in this sequence) that update all the edges following one shortest path from $s$ to $v$, then after that we will have $dist[v] = distance(s,v)$.

But we do not know the the shortest path in advance. That's fine. As in graphs without negative cycles the number of edges in the (simple) shortest path will not exceed $(|V|-1)$, the Bellman-Ford algorithm simply update *all* edges in each round, and do this $(|V|-1)$ times. This therefore guarantees that the $i$-th edge on the shortest path can be updated during the $i$-th round. Consequently, this guarantees the existence of a sequence of update procedures that update all edges following the shortest path. This analysis leads to the following conclusion, which proves the correctness of Bellman-Ford algorithm. See an illustration in Figure 2.

**Fact 7.** If $G$ does not contain negative cycle, then we have $dist[v] = distance(s,v)$ for every $v \in V$ after $|V|-1$ rounds of updates. In particular, let $p$ be one shortest path from $s$ to $v$ with $k$ edges. Then after $k$ rounds of updates, $dist[v] = distance(s,v)$.

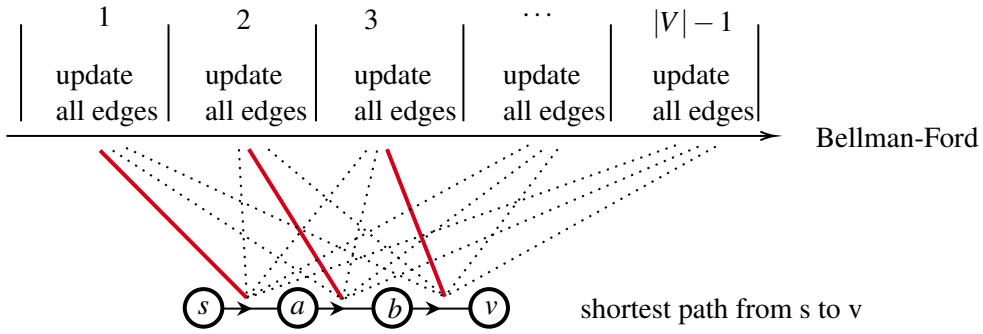



Figure 2: Illustration of the correctness of the Bellman-Ford algorithm. Dotted lines represent additional updates on the corresponding edge.

## Detecting Negative Cycles

We can slightly modify Bellman-Ford algorithm to detect if a given graph contains negative cycle that is reachable from $s$. The algorithm does one more round of updates, in which it determines if some *dist* value can be further reduced.

Algorithm Bellman-Ford-Detect-Negative-Cycle ($G = (V,E)$, $l(e)$ for any $e \in E$, $s \in V$)
    init an array *dist* of size $|V|$;
    $dist[s] = 0$; $dist[v] = \infty$ for any $v \neq s$;
    for $k = 1 \rightarrow |V| - 1$
        for each edge $(u,v) \in E$
            $update(u,v)$;
        end for;
    end for;
    for each edge $(u,v) \in E$
        if ($dist[v] > dist[u] + l(u,v)$): report $G$ contains negative cycle and exit
    end for;
    report that $G$ does not contain negative cycle
end algorithm;

Let's show that above algorithm is correct. We first prove that, if $G$ does not contain negative cycle (reachable from $s$), then in above additional round $dist[v] > dist[u] + l(u,v)$ will never happen, i.e., we will get the report that "$G$ does not contain negative cycle". As per Fact 7 and the assumption that $G$ does not contain negative cycle, we know that $dist[v] = distance(s,v)$ after $|V| - 1$ rounds. Also, according to Fact 1, update function will never make $dist[v]$ smaller than $distance(s,v)$ when $G$ does not contain negative cycle. Hence, during the $|V|$-th round in above algorithm, none of the *dist* value can be further reduced.

We then prove that, if $G$ contains negaive cycle (reachable from $s$), then in above additional round, there must exist an edge $(u,v)$ such that $dist[v] > dist[u] + l(u,v)$. Suppose conversely that, in above additional round, all edges satisfy $dist[v] \leq dist[u] + l(u,v)$. Let $C = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{k-1} \rightarrow v_k \rightarrow v_1$ be one negative reachable from $s$. We have $\sum_{e \in C} l(e) < 0$ as $C$ is a nagative cycle. Applying $dist[v] \leq dist[u] + l(u,v)$ to all edges in $C$ gives:

$$\begin{array}{rcl} dist[v_2] & \leq & dist[v_1] + l(v_1,v_2) \\ dist[v_3] & \leq & dist[v_2] + l(v_2,v_3) \\ & \cdots & \\ dist[v_k] & \leq & dist[v_{k-1}] + l(v_{k-1},v_k) \\ dist[v_1] & \leq & dist[v_k] + l(v_k,v_1) \end{array}$$

Summing up both sides of all above inequalities gives $\sum_{e \in C} l(e) \geq 0$, a contradiction.

## Shortest Path of DAGs

Let $G = (V,E)$ be a DAG with possibly negative edge length $c(e)$ for each $e \in E$. We want to find the distance from a given source $s$ to each vertex $v \in V$. Note that $G$ does not contain negative cycles simply because $G$ does not contain any cycle. We can design a simplified version of Bellman-Ford algorithm to solve this problem. In particular, we only need to do one round of "update" (instead of $|V| - 1$ rounds as in Bellman-Ford algorithm). But, the edges cannot be updated in a arbitrary order. In fact, we first need to find a linearization of $G$ (recall that a directed graph can be linearized if and only if it is a DAG), and then update

the in-edges of vertices following this linearization.

```
Algorithm BF-DAG (DAG G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init array dist of size |V|: dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    calculate a linearization of G;
    for v ∈ V following the order of linearization
        for each (u,v) ∈ E;
            update(u,v);
        end for;
    end for;
    for each v ∈ V: report: distance(s,v) = dist[v];
end algorithm;
```

The correctness of above algorithm for DAGs can also be proved in the same way as in the Bellman-Ford algorithm. For any vertex $v$, let $p = s \to v_1 \to v_2 \to \cdots \to v$ be one shortest path from $s$ to $v$; we have proved that (for Bellman-Ford algorithm), if the run of the algorithm contains a sequence of updates that sequentially update the edges in this shortest path $p$, then $dist[v]$ will be equal to $distance(s,v)$ after these update procedures. The above algorithm update the in-edges of vertices sequentially following a linearization $X$. By the definition of linearization, the list of vertices of any path, including $p$, must be a *subsequence* of $X$ (i.e., $s$ is before $v_1$ in $X$, $v_1$ is before $v_2$ in $X$, etc). Therefore, edges in $p$ must be updated sequentially by the above algorithm, i.e., edge $(s,v_1)$ gets updated when processing $v_1$, edge $(v_1,v_2)$ gets updated when processing $v_2$, and so on. See Figure 3 for an example.

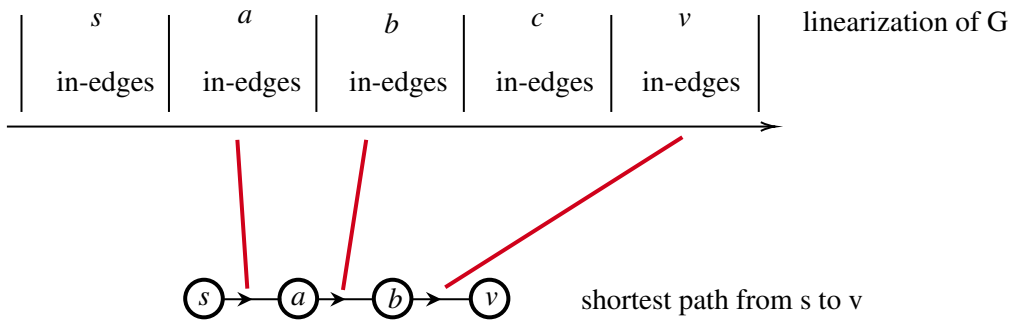



Figure 3: Illustration of the correctness of the above algorithm. Assume $s \to a \to b \to v$ be one shortest path from $s$ to $v$ in $G$. Then in any linearization of $G$, $s$ is before $a$, $a$ is before $b$, and $b$ is before $v$. So, edge $(s,a)$ will be updated when processing the in-edges of $a$, $(a,b)$ will be updated when processing the in-edges of $b$, and $(b,v)$ will be updated when processing the in-edges of $v$. Therefore, there exists a sequence of updates that sequentially update these 3 edge.

# Lecture 18 Shortest Path Tree

So far we have designed three algorithms, BFS, Dijkstra's algorithm, and Bellman-Ford algorithm to solve shortest path problems for unit edge length, positive edge length, and possible negative edge length. In these algorithm, $dist[v]$ will give the *length* of the shortest path from $s$ to $v$. How to find the actual shortest path? Before designing algorithm to find paths (we will do it by modifying all three algorithm), let's think about how to store them first. Recall that we want to store $|V|$ paths, ones from $s$ to each vertex in $V$. If we explicitly store these $|V|$ paths in a naive way, it may take $O(|V|^2)$ space. Can we do better?

In fact, we can use linear space to store these $|V|$ shortest paths from a single source $s$. How is that possible? The reason is the *optimal substructure* property that the shortest path problem satisfies. Assume $s = v_0 \to v_1 \to v_2 \cdots v_k$ be the shortest path from $s$ to $v_k$. Then $s \to v_i$, for any $0 \le i < k$, is also the one shortest path from $s$ to $v_i$. This implies that storing the path from $s$ to $v_k$ is enough to represent the shortest paths from $s$ to each $v_i$.

But one such path is not enough to represent all shortest paths from $s$. In fact, all shortest paths can be represented as a tree, called *shortest path tree*. Without loss of generality, assume that source $s$ can reach all vertices in $V$. We say $T$ is a *shortest path tree* of $G$ w.r.t. source vertex $s$ if:

1. $T$ is a rooted tree and its root is $s$;
2. vertices of $T$ is $V$;
3. edges of $T$ is a subset of $E$;
4. for every $v \in V$, the unique path from $s$ to $v$ in $T$ is one shortest path from $s$ to $v$ in $G$.

We will show later that, a shortest path tree always exists. Such tree may not be unique though (see Figure 1). As in the shortest path tree, each vertex, except $s$, has exactly one in-edge, we can use an array *prev* of size $|V|$ to store this tree. Array *prev* is indexed by the vertices, and for each $v \in V$, $prev[v]$ stores the parent of $v$ in the tree (see Figure 1). Such array, which takes linear space, therefore represents the shortest path from $s$ to every vertex.

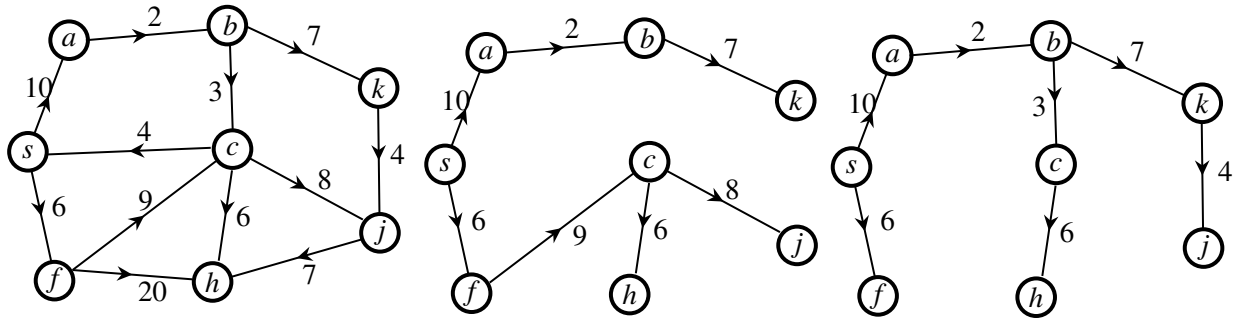



Figure 1: A directed graph and its two shortest path trees. The array representation for the first tree is $prev[s,a,b,c,f,h,j,k] = [null,s,a,f,s,c,c,b]$.

We now modify the three algorithms, BFS, Dijkstra's algorithm, and Bellman-Ford algorithm, to allow them to generate a shortest path tree. The idea is that, whenever $dist[v]$ gets updated as $dist[u] + l(u,v)$ we also immediately assign $prev[v]$ as $u$. See their pseudo-codes below.

```
Algorithm BFS (G = (V,E), s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    prev[v] = null, for any v ∈ V;
    init an empty queue Q;
    dist[s] = 0;
    insert (Q,s);
    while (empty (Q) = false)
        u = find-earliest (Q);
        delete-earliest (Q);
        for each edge (u,v) ∈ E
            if (dist[v] = ∞)
                dist[v] = dist[u] + 1;
                prev[v] = u;
                insert (Q, v);
            end if;
        end for;
    end while;
end algorithm;
```

```
Algorithm Dijkstra (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    dist[v] = ∞, for any v ∈ V;
    prev[v] = null, for any v ∈ V;
    init an empty priority queue PQ;
    for any v ∈ V: insert (PQ, v), where the priority of v is ∞;
    dist[s] = 0;
    decrease-key (PQ,s,0);
    while (empty (PQ) = false)
        u = find-min (PQ);
        delete-min (PQ);
        for each edge (u,v) ∈ E
            if (dist[v] > dist[u] + l(u,v))
                dist[v] = dist[u] + l(u,v);
                prev[v] = u;
                decrease-key (PQ, v, dist[v]);
            end if;
        end for;
    end while;
end algorithm;
```

```
Algorithm Bellman-Ford (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init an array dist of size |V|;
    dist[s] = 0; dist[v] = ∞ for any v ≠ s;
    prev[v] = null, for any v ∈ V;
    for k = 1 → |V| − 1
        for each edge (u,v) ∈ E
            update(u,v);
        end for;
    end for;
end algorithm;

procedure update (edge (u,v) ∈ E)
    if (dist[v] > dist[u] + l(u,v))
        dist[v] = dist[u] + l(u,v);
        prev[v] = u;
    end if;
end procedure;
```

The resulting *prev* array corresponds to a graph called $T$, in which the vertices are $V$ and edges are $\{(prev[v], v) \mid v \in V\}$. We now prove that this graph is indeed one shortest path tree, by showing $T$ satisfying the 4 conditions in the definition. It is trivial to verify that the conditions (2) is satisfied, as the vertices are $V$; condition (3) is easy as well, as in any of the algorithm, assigning $u$ to $prev[v]$ already happens when examining edge $(u,v) \in E$. Below we then prove condition (1) and (4).

**Fact 1.** At any time of the algorithm, for any $v \in V$, let $u = prev[v]$, if $u \neq null$ then $dist[v] = dist[u] + l(u,v)$; in particular, when the algorithm terminates, we have $distance(s,v) = distance(s,u) + l(u,v)$.

The first part of above fact is a direct consequence of each algorithm as we update *prev* right after updating $dist[v]$ as $dist[u] + l(u,v)$. The secon part also uses the correctness of each algorithm, i.e., when each algorithm terminates, we have $dist[v] = distance(s,v)$.

We now prove condition (1), i.e., $T$ is a tree with root being $s$ (although we use $T$ but we have not showed that it is a tree). Since $T$ corresponds to array *prev*, it is obvious that in-degree of $s$ is 0 and that each vertex in $T$, except $s$, has in-degree of 1. We therefore just need to prove that $T$ does not contain cycle. We need a stronger assumption to make it true. That is, the graph $G$ cannot contain negative nor length-zero cycles.

**Fact 2.** $T$ does not contain cycle, if all cycles in $G$ are of positive length.

*Proof.* Suppose conversely that there exists a cycle $C = v_1 \to v_2 \to \cdots \to v_k \to v_1$ in $T$. Following above fact, we have $dist[v_i] = dist[v_{i-1}] + l(v_{i-1}, v_i)$, for every $1 < i \leq k$ and $dist[v_1] = dist[v_k] + l(v_k, v_1)$. Summing up both sides gives $\sum_{e \in C} l(e) = 0$, contradicting to the the assumption that all cycles have positive length. □

Last, we prove condition (4):

**Claim 1.** For every $v \in V$, the unique path from $s$ to $v$ in $T$ is one shortest path from $s$ to $v$ in $G$, if all cycles in $G$ are positive cycles.

*Proof.* Let $p : s = v_0 \to v_1 \to v_2 \to \cdots \to v_k$ be the path from $s$ to $v_k$ in $T$. By the construction of $T$, we know that $prev[v_i] = v_{i-1}$, $1 \leq i \leq k$. Therefore, using Fact 1, we have $distance(s,v_i) = distance(s,v_{i-1}) + l(v_{i-1}, v)$, for each $i = 1, 2, \cdots, k$. We sum up the left side and the right side of these $k$ equations. Canceling

out $distance(s, v_i)$, $i = 1, 2, \cdots, k-1$, on both sides gives $distance(s, v_k) = distance(s, s) + \sum_{i=1}^{k} l(v_{i-1}, v_i) = \sum_{i=1}^{k} l(v_{i-1}, v_i)$, implying that the length of $p$ is equal to the distance from $s$ to $v_k$, i.e., it is one shortest path from $s$ to $v_k$. □

# Lecture 19 Tramp Steamer Problem

This is about exercise 4.22, on page 125 of book *Algorithms* (read it for a real-world application from which the formulation below is abstracted). Formally, we are given a directed graph $G = (V, E)$, where each vertex $v \in V$ is associated with a *profit* $p(v) > 0$, and each edge $e \in E$ is associated with a *cost* $c(e) > 0$, and we seek a simple (i.e., each vertex appears at most once) cycle $C$ of $G$ such that the profit-cost ratio of $C$, defined as $r(C) := (\sum_{v \in C} p(v))/(\sum_{e \in C} c(e))$ is maximized.

Let $r^* = \max_C r(C)$ be the maximized ratio, and let $C^* = \arg\max_C r(C)$ be the corresponding optimal cycle. In fact, it's hard to calculate the optimal solution (i.e., $C^*$ and $r^*$). Instead, we will approximate it, but with *any given accuracy*. Specifically, for any given $\varepsilon > 0$ (note: $\varepsilon$ can be arbitrarily small), we will determine a cycle $C$ such that $r(C) < r^* \leq r(C) + \varepsilon$. In other words, we seek an interval of length $\varepsilon$ that contains the optimal ratio.

## Binary Search

We will use *binary search* to solve above problem. To do it, we need two components:

1. An algorithm to decide if a given ratio $r$ is a lower bound of $r^*$ (i.e., $r < r^*$). In other words, to decide if $r$ is achievable (i.e., if there exists a cycle $C$ such that $r(C) > r$). We will design this algorithm; its interface will be decide-lower-bound ($G$, $r$), which returns true if $r < r^*$ and returns false if $r \geq r^*$.
2. Initial bounds $r_1$ and $r_2$ for $r^*$ to start search from, i.e., $r_1 < r^* \leq r_2$.

We will resolve about two components later on. Suppose they are available, we can design the following binary search algorithm. This algorithm, called binaray-search ($G$, $a$, $b$), is recursive. Whenever it is called, it is guaranteed that $a < r^* \leq b$.

Algorithm binary-search ($G = (V, E)$, $a$, $b$)
    if $b - a \leq \varepsilon$: report interval $(a, b]$ and exit;
    let $m = (a + b)/2$;
    let $z$ = decide-lower-bound ($G$, $m$);
    if $z$ = true: return binary-search ($G$, $m$, $b$);
    else: return binary-search ($G$, $a$, $m$);
end algorithm;

With available bounds $r_1$ and $r_2$, we can call binary-search ($G$, $r_1$, $r_2$), which will return the desired interval of length at most $\varepsilon$ that includes $r^*$.

## Algorithm to Decide Lower Bound

We now design an algorithm to decide if a given ratio $r$ is a lower bound of $r^*$. Note that $r < r^*$ if and only if there exists a cycle $C$ such that $r(C) > r$. (Proof: if $r < r^*$, then the optimal cycle $C^*$ satisfies that $r^* = r(C^*) > r$ and hence such $C = C^*$ exists; if there exists $r(C) > r$ then $r^* \geq r(C) > r$.)

We can write

$$\begin{aligned} & r < r^* \\ \iff & \exists \text{ cycle } C \text{ s.t. } r(C) = (\textstyle\sum_{v \in C} p(v))/(\sum_{e \in C} c(e)) > r \\ \iff & \exists \text{ cycle } C \text{ s.t. } r \cdot \textstyle\sum_{e \in C} c(e) - \sum_{v \in C} p(v) < 0 \end{aligned}$$

$$\iff \quad \exists \text{ cycle } C \text{ s.t. } \sum_{e\in C} r\cdot c(e) + \sum_{v\in C}(-p(v)) < 0$$

We transform the problem of deciding the last condition into the problem of deciding negative-cycle. Given $G=(V,E)$ and $r$, we build a new graph $G_1=(V_1,V_2)$. See Figure 1. Each vertex $v$ in $G$ will be replaced by two vertices $v_1$ and $v_2$ in $G_1$, and $v_1$ and $v_2$ will be connected by an new edge $(v_1,v_2)$; this new edge will have a length of $-p(v)$. Each edge $e=(u,v)$ in $G$ will be replaced by edge $(u_2,v_1)$ in $G_1$; this edge will have a length of $r\cdot c(e)$. Clearly, based on how $G_1$ is constructed, we have:

1. There exists a cycle $u\to v\to w\to\cdots\to u$ if and only if there exists a cycle $u_2\to v_1\to v_2\to w_1\to w_2\to\cdots\to u_1\to u_2$ in $G_1$.

2. Let $C$ be a cycle in $G$ and let $C_1$ be its corresponding cycle in $G_1$. Then the length of $C_1$ in $G_1$ is exactly $\sum_{e\in C} r\cdot c(e)+\sum_{v\in C}(-p(v))$.

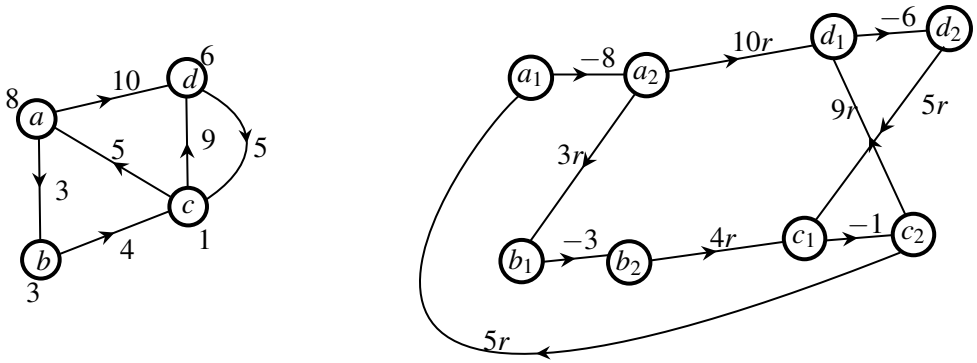



Figure 1: Constructing $G_1$ from $G$ and $r$.

Hence, we can keep writing above reasoning:

$$\begin{aligned}
& r < r^* \\
\iff\ & \exists \text{ cycle } C \text{ in } G \text{ s.t. } \textstyle\sum_{e\in C} r\cdot c(e) + \sum_{v\in C}(-p(v)) < 0 \\
\iff\ & \exists \text{ cycle } C_1 \text{ in } G_1 \text{ s.t. the length of } C_1 \text{ is smaller than } 0 \\
\iff\ & G_1 \text{ contains negative cycle}
\end{aligned}$$

We therefore successfully transformed the problem of deciding if $r<r^*$ into a problem of deciding if $G_1$ contains negative cycle, which we know how to solve. The pseudo-code is given below.

Algorithm decide-lower-bound ($G=(V,E)$, $r$)
  build $G_1=(V_1,E_1)$;
  call an algorithm to decide if $G_1$ contains negative cycle;
  if $G_1$ contains negative cycle: return true (i.e., $r<r^*$);
  else: return false (i.e., $r\geq r^*$);
end algorithm;

The running time of above algorithm is dominated by the procedure of deciding negativ cycles, which is $O(|V_1|\cdot|E_1|)$. Notice that $|V_1|=2|V|$ and $|E_1|=|V|+|E|$. Hence, the running time of above algorithm is $O(|V|^2+|V|\cdot|E|)$.

## Initial Bounds

We now find an initial bounds $r_1$ and $r_2$ for $r^*$. We can use a trivial lower bound, i.e., $r_1 = 0$. What is a possible upper bound of $r^*$? Notice that $r(C)$ is the ratio of the sum of vertex-profits and the sum of edge-costs, and the number of vertices and the number of edges in a cycle are the same. Hence, we have an upper bound: $r_2 = \max_{v \in V} p(v) / \max_{e \in E} c(e)$. A better upper bound can be obtained by realizing that in any cycle edge $e = (u,v)$ is always followed by vertex $v$. Hence, we have a better bound: $r_2 = \max_{e=(u,v) \in E} p(v)/c(e)$.

## Running Time Analysis

To see the running time, assume $t$ is the number of iterations in the binary search. We have $(r_2 - r_1)/2^t = \varepsilon$. Hence we have $t = \log(r_2/\varepsilon)$. The overall running time is therefore $O(\log(r_2/\varepsilon)(|V|^2 + |V||E|))$. Notice that this running time is a function of $\varepsilon$, as expected. Notice too, that this running time involves numeric value, i.e., $r_2/\varepsilon$. The *input-size* of any numeric value $x$ is $\log x$. Therefore $\log(r_2/\varepsilon)$ is linear in the input-size of $r_2/\varepsilon$. Hence, the entire algorithm runs in polynomial-time.

To comapre, consider the linear-search algorithm given below.

Algorithm linear-search ($G = (V,E)$, $a$, $b$)
    init $r = 0$;
    let $z$ = decide-lower-bound ($G$, $r$);
    if $z$ = true: $r = r + \varepsilon$;
    else: return $(r - \varepsilon, r]$;
end algorithm;

This algorithm is correct, but it may call decide-lower-bound $r_2/\varepsilon$ times. So its overall running time is $O(r_2/\varepsilon(|V|^2 + |V||E|))$. This is *not* a polynomial-time algorithm, as $r_2/\varepsilon$ is exponential in its input-size, which is $\log(r_2/\varepsilon)$.

In conclusion, binary-search is essential to achieve an efficient algorithm for this problem.

# Lecture 20 Longest Increasing Subsequences

## Framework of Dynamic Programming Algorithm

A typical dynamic programming algorithm consists of following steps:

1. Define subproblems (or equivalently, partition the search space into sub-spaces), $P_1, P_2, \cdots, P_n$; solving all these subproblems will solve the original problem.
2. Develop a recursion for these subproblems. Formally, let $opt(P_k)$ be the optimal solution of $P_k$. A recursion is a function that calculates $opt(P_{k+1})$ using $opt(P_1), opt(P_2), \cdots, opt(P_k)$.
3. Fill a *dynamic programming table*, in which each entry stores the optimal solution of a subproblem, using above recursion.

## Optimal Substrcture Property

Let $P_1, P_2, P_3, \cdots, P_n$ be a series of subproblems defined for original problem $P$. If we have that, the optimal solution of $P_k$, denoted as $opt(P_k)$, can be calculated solely from the *optimal solution* of previous subproblems, i.e., $opt(P_k) = f(opt(P_1), opt(P_2), \cdots, opt(P_{k-1}))$, then we say problem $P$ satisfies the optimal substructure property. Intuitively, the optimal substructure property states that the optimal solution of a larger problem can be *assembled* using the optimal solutions of the smaller subproblems.

How to see if a problem satisfies the optimal substructure property? Usually we can verify if its opporsite side is true, i.e., if the optimal solution of a bigger subproblem *implies* the optimal solution of the smaller subproblems. Take the shortest path problem as an example we showed before: Let $p = s \to w \to \cdots \to u \to v$ be the shortest path from $s$ to $v$. Then clearly its sub-path $s \to w \to \cdots \to u$ is the shortest path from $s$ to $u$ (because, if there exists a shorter path from $s$ to $u$ then we can find a shorter one from $u$ to $v$ than $p$). This suggests that the shortest path problem satisfies the optimal substructure property. We will use more examples to demonstrate it and how to determine a proper definition accordingly.

## Longest Increasing Subsequences

Let $A[1 \cdots n]$ be an array with $n$ distinct integers. A *subsequence* of array $A$ is a subset of $A$ but follows the same order. For example, $(A[2], A[4], A[5])$ is a subsequence of $A$ while $(A[3], A[1], A[4])$ is not. An *increasing subsequence* of $A$ means the values of the the subsequence are increasing. Formally, an increasing subsequence of length $k$ can be written as $(A[i_1], A[i_2], \cdots, A[i_k])$ where $1 \le i_1 < i_2 < \cdots < i_k \le n$, and $A[i_1] < A[i_2] < \cdots < A[i_k]$. The *longest increasing subsequence problem* is, given array $A[1 \cdots n]$, to seek an increasing sequence with largest length.

For example, let $A = [1, 5, 3, 8, 9, 4, 7, 6]$. Then one of the longest increasing sequence is $[1, 3, 4, 7]$. Notice that, the optimal solution may not be unique. In this example, you may find more increasing sequences of length 4.

We want to design a dynamic programming algorithm to solve this problem. Recall that, in order to give a proper definition of subproblems, we can try "reverse engineering", i.e., starting from the optimal solution, trying to determine if it *implies* the optimal solution of certain subproblem. If it does, then that subproblem is likely the one we could use in designing the DP algorithm.

Let's try this approach with above example. We know that $[1, 3, 4, 7]$ is one optimal solution. Does its substructure, $[1, 3, 4]$, give the optimal solution of certain subproblem? A natural guess would be that, $[1, 3, 4]$, is the longest increasing subsequence of $A[1 \cdots 6] = [1, 5, 3, 8, 9, 4]$. But, a further look denies this

conjecture, as $[1,3,8,9]$ is a longer increasing subsequence of $A[1\cdots 6]$. Let's have another try. We still consider $A[1\cdots 6]$, but require that the last number of the increasing subsequence must be $A[6]=4$. (Note: this is again a commonly-used technique, i.e., introducing a variable or a constraint into the definition of the subproblems.) Among all such increasing subsequences of $A[1\cdots 6]$, is $[1,3,4]$ the optimal one? Yes! Why? Again we can prove by contradiction: suppose that there is a longer increasing subsequence of $A[1\cdots 6]$ that ends at $A[6]=4$, we can concatenate it with $A[7]=7$ to obtain a longer, than $[1,3,4,7]$, increasing subsequence of $A$, contradicting to the optimality of $[1,3,4,7]$.

We summarize above example with the following optimal substructure statement.

**Fact 1.** If $(A[i_1],A[i_2],\cdots,A[i_k])$ be the longest increasing subsequence of $A[1\cdots i_k]$, then among all increasing subsequences of $A[1\cdots i_{k-1}]$ with last number being $A[i_{k-1}]$, $(A[i_1],A[i_2],\cdots,A[i_{k-1}])$ is (one of) the longest one.

Above property directly suggests the definition of subproblems. We define $opt(k)$ as the longest increasing subsequence of $A[1\cdots k]$ with $A[k]$ being the last number, and define $L(k)$ as the length of $opt(k)$. In other words, among all increasing subsequences of $A[1\cdots k]$ with last number being $A[k]$, $opt(k)$ is the optimal solution and $L(k)$ is its length.

We defined $n$ subproblems. Suppose we can solve all of them. What's the longest increasing subsequence of $A$? Clearly, that's $\max_{1\le k\le n} L(k)$.

We now develop a recursion. How to calculate $opt(k)$? Again we enumerate the last step. By definition of the subproblem, $A[k]$ is the last number of $opt(k)$. We therefore enumerate its second last number, which must be one of $A[j]$ satisfying $1\le j<k$ and $A[j]<A[k]$. We don't know which one is correct so we enumerate all possibilities. (And once we enumerate, we can assume we know it.) Suppose that $A[j]$ is the second last number of $opt(k)$, then according to Fact 1, $opt(k)$ can be obtained by calculating $opt(j)$ followed by $A[k]$. Formally, the recursion is:

$$L(k)=1+\max_{1\le j<k, A[j]<A[k]} L[j].$$

We give the pseudo-code below. We maintain two arrays, $L(k)$, to store the length of the $opt(k)$, and $P(k)$, to maintain the tracking-back pointers, $1\le k\le n$. The optimal solution $opt(k)$ can be constructed by using $T$. The running time of the algorithm is $O(n^2)$.

```
Algorithm longest-increasing-sequence (array A[1···n])
    for k = 1 → n
        L(k) = 1;
        T(k) = 0;
        for j = 1 → k − 1
            if (A[j] < A[k] and L(j) + 1 > L(k))
                L(k) = L(j) + 1;
                T(k) = j;
            end if;
        end for;
    end for;
    report: max_{1≤k≤n} L(k) is the length of the longest increasing subsequence of A;
end algorithm;
```

# Lecture 21 Edit Distance Problem

The edit distance problem is largely motivated by and abstracted from genome evolution. There are three types of nucleotide-level mutations, substitutions, insertions, and deletions. See Figure 1. Given two DNA sequences, for examples, the sequences of two homologous genes, we want to infer the most likely mutation events happened between them during the evolutionary history. This can be modeled as the following edit distance problem: given two strings $X$ and $Y$ over alphabet $\Sigma$, to find the minimum number of such events that transforms $X$ into $Y$. We denote by $d(X,Y)$ as the edit distance between $X$ and $Y$. See Figure 2 for an example, where we can write $d(X,Y) = 4$.

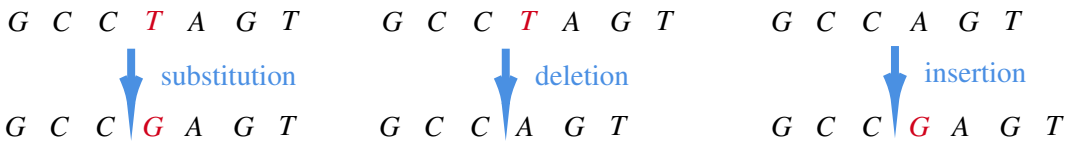



Figure 1: Illustration of the 3 types of mutation events in genome evolution.

An equivalent and convenient way to represent edits between two sequences is *alignment*. See Figure 2. In an alignment, deletions and insertions are represented with *gaps*, shown with $-$. (Note that deletions on $X$ is equivalent to insertions on $Y$; deletions on $Y$ is equivalent to insertions on $X$.) Both exactly-matched characters and mismatched-characters (i.e., requires a substitutions event), are shown by vertical-bars with substitutions highlighted. Clearly, there is a one-to-one correspondence between a series of events that transforms $X$ into $Y$ and an alignment between $X$ and $Y$, and the number of events is equal to the number of gaps and mismatches in the alignment. Hence, calculating edit distance between $X$ and $Y$ is equivalent to seeking the optimal alignment between $X$ and $Y$ (here, "optimal" means the number of gaps and mismatches in the alignment is minimized).

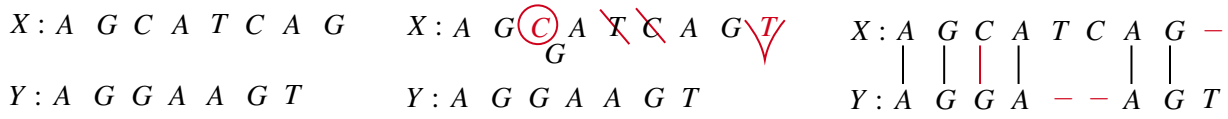



Figure 2: Illustration of the edit distance and the corresponding optimal alignment.

## A Dynamic Programming Algorithm

We now design a dynamic programming to find the optimal alignment of two given strings. We first look into the optimal substructures. Suppose that $\mathcal{A}$ is the optimal alignment of two strings $X$ and $Y$ (see Figure 3). Then clearly its substructure, i.e., the first $k$ columns, gives the optimal alignment between the corresponding two substrings, which can be obtained by removing the gaps in the first $k$ columns of $\mathcal{A}$. (Proof: if the two substrings can be aligned with a better alignment (with smaller number of gaps and mismatches), then we can have a better alignment, than $\mathcal{A}$, between $X$ and $Y$, by concatenating to it the remaining columns of $\mathcal{A}$.)

So, the alignment problem indeed satisfies the optimal substructure property. And the substructure suggests defining the subproblems as to computing the optimal alignment between a substring of $X$ (starting from beginning) and a substring of $Y$ (starting from beginning). Formally, let $F(i,j)$ be the edit distance between $X[1\cdots i]$ and $Y[1\cdots j]$. Clearly, we have $d(X,Y) = F(|X|,|Y|)$. Now we develop a recursion. In order to calculate the optimal alignment between $X[1\cdots i]$ and $Y[1\cdots j]$, consider the possible scenario of the last step, i.e., how $X[i]$ and $X[j]$ are placed in the optimal alignment. There are three cases. See Figure 4.

First, $X[i]$ is aligned to $Y[j]$ in the optimal alignment. In this case, the optimal alignment between $X[1\cdots i]$


Figure 3: Illustration of the optimal substructure of alignment. The left figure shows the optimal alignment between $X$ and $Y$. Then we can remove any number of columns from right-side, and the resulting alignment must be the optimal alignment of the corresponding two substrings. For examples, the middle figure shows the optimal alignment between $X' = AGCATCAG$ and $Y' = AGGAAG$, where one column of the alignment in the left-figure is removed; and the right figure shows the optimal alignment between $X'' = AGCATC$ and $Y'' = AGGA$, where three columns of the alignment in the left-figure is removed.

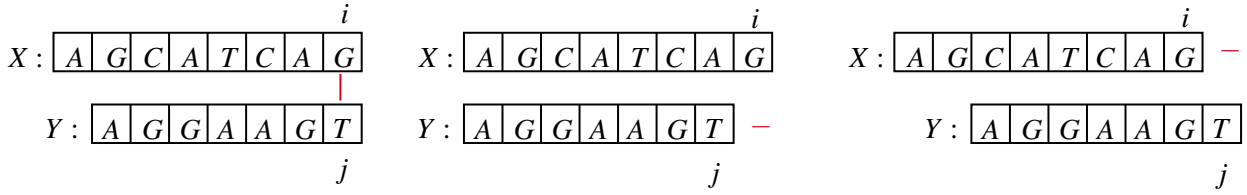



Figure 4: Illustration of the three cases in developing the recursion.

and $Y[1\cdots j]$ can be obtained by concatenating the optimal alignment between $X[1\cdots i-1]$ and $Y[1\cdots j-1]$ and this column of $(X[i],Y[j])^T$. Hence, in this case, $F(i,j) = F(i,j) + \delta(X[i] \neq Y[j])$, where the delta function $\delta(X[i] \neq Y[j]) = 1$ if $X[i] \neq Y[j]$, i.e., a mismatch that requires a substitution event, and $\delta(X[i] \neq Y[j]) = 0$ if $X[i] = Y[j]$, i.e., an exactly-match that does not need any event.

Second, $X[i]$ is not aligned (or, aligned to a gap) in the optimal alignment. In this case, the optimal alignment between $X[1\cdots i]$ and $Y[1\cdots j]$ can be obtained by concatenating the optimal alignment between $X[1\cdots i-1]$ and $Y[1\cdots j]$ and this column of $(X[i],-)^T$. Hence, in this case, $F(i,j) = F(i-1,j)+1$, where the extra 1 represents a deletion on $X$ is needed.

Third, $Y[j]$ is not aligned (or, aligned to a gap) in the optimal alignment. In this case, the optimal alignment between $X[1\cdots i]$ and $Y[1\cdots j]$ can be obtained by concatenating the optimal alignment between $X[1\cdots i]$ and $Y[1\cdots j-1]$ and this column of $(-,Y[j])^T$. Hence, in this case, $F(i,j) = F(i,j-1)+1$, where the extra 1 represents a deletion on $Y$ is needed.

Combined, the recursion is given below.

$$F(i,j) = \min \begin{cases} F(i-1,j-1) + \delta(X[i] \neq Y[j]) \\ F(i-1,j)+1 \\ F(i,j-1)+1 \end{cases}$$

We now can complete the algorithm. The algorithm fills a DP table. See Figure 5. To facilitate the recursion, we add a new row and a new column in the table, and fill them in the initialization step. We then fill the table, either row by row, or column by column (why both work?). The right bottom entry, $F(|X|,|Y|)$, gives the edit distance between $X$ and $Y$. The actual alignment can be constructed by maintaining a tracking back pointer on each entry. The running time of the algorithm is therefore $O(|X||Y|)$.

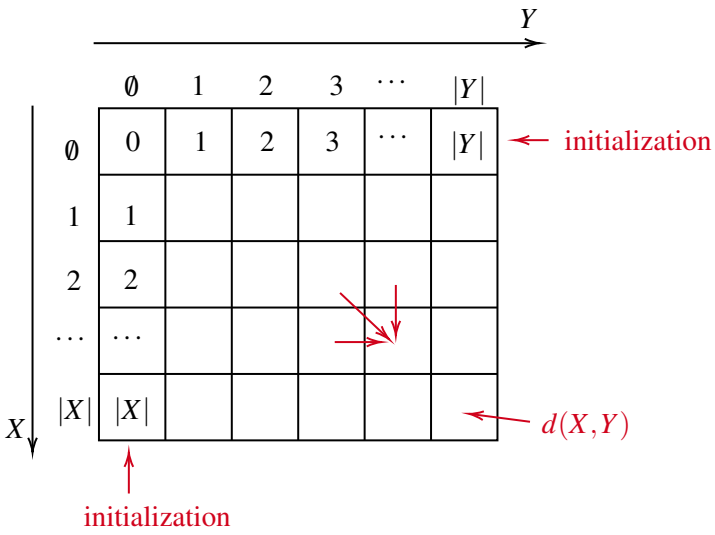



Figure 5: The dynamic programming table.

Algorithm edit-distance (two strings $X$, $Y$)
  $F(0,j) = j$, for any $0 \le j \le |Y|$
  $F(i,0) = i$, for any $0 \le i \le |X|$
  for $i = 1 \to |X|$
    for $j = 1 \to |Y|$
      $F(i,j) = \min\{F(i-1,j-1) + \delta(X[i] \neq Y[j]), F(i-1,j)+1, F(i,j-1)+1\}$;
    end for;
  end for;
  report: $F(|X|,|Y|) = d(X,Y)$;
end algorithm;

## Transforming into Shortest Path Problem

Another way to solve the edit distance problem is to transform it into a shortest-path problem. See Figure 6. We build a graph with $(|X|+1)(|Y|+1)$ vertices, $v_{ij}$, $0 \le 0 \le |X|$ and $0 \le j \le |Y|$. (Imagine placing a vertex in each entry of above DP table.) There are three in-edges for vertex $v_{ij}$, when $i \ge 1$ and $j \ge 1$, corresponding to the 3 cases of the recursion. All vertical edges have edge length of 1; all horizontal edges have edge length of 1; all diagonal edges have edge length of either 1 or 0: if $X[i] \neq Y[j]$ then the edge $(v_{i-1,j-1}, v_{ij})$ has a length of 1, and otherwise this edge has a length of 0.

Clearly, there is a one-to-one correspondence between a path from $v_{00}$ to $v_{|X|,|Y|}$ and an alignment between $X$ and $Y$, and the length of the path equals to the number of operations in the alignment. See Figure 7. In fact, a path from $v_{00}$ to $v_{|X|,|Y|}$ suggests how to align $X$ and $Y$: a diagonal edge pointing to vertex $v_{ij}$ implies aligning $X[i]$ and $Y[j]$; a vertical edge pointing to vertex $v_{ij}$ implies not to aligning $X[i]$ (or aligning $X[i]$ with gap); a horizontal edge pointing to vertex $v_{ij}$ implies not to aligning $Y[j]$ (or aligning $Y[j]$ with gap). You can also see that the length of this path equals to the number of gaps and mismatches in the resulting alignment. This correlation is bi-directional: an alignment can be mapped into a path in the graph too.

The above analysis leads to that, the optimal alignment between $X$ and $Y$ corresponds to the shortest path

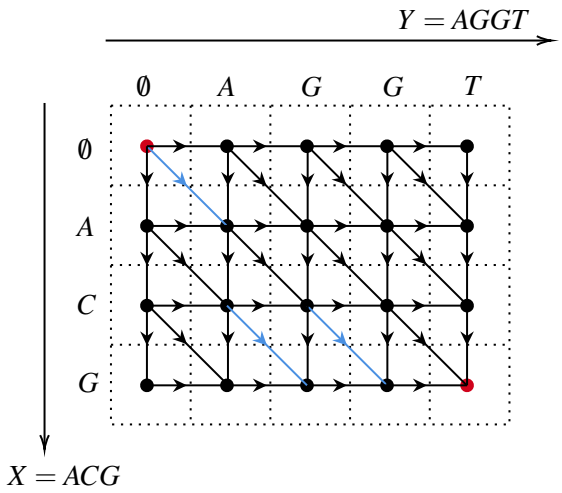



Figure 6: The graph constructed for $X = ACG$ and $Y = AGGT$. The 3 blue edges have length of 0, and all other edges have length 1.

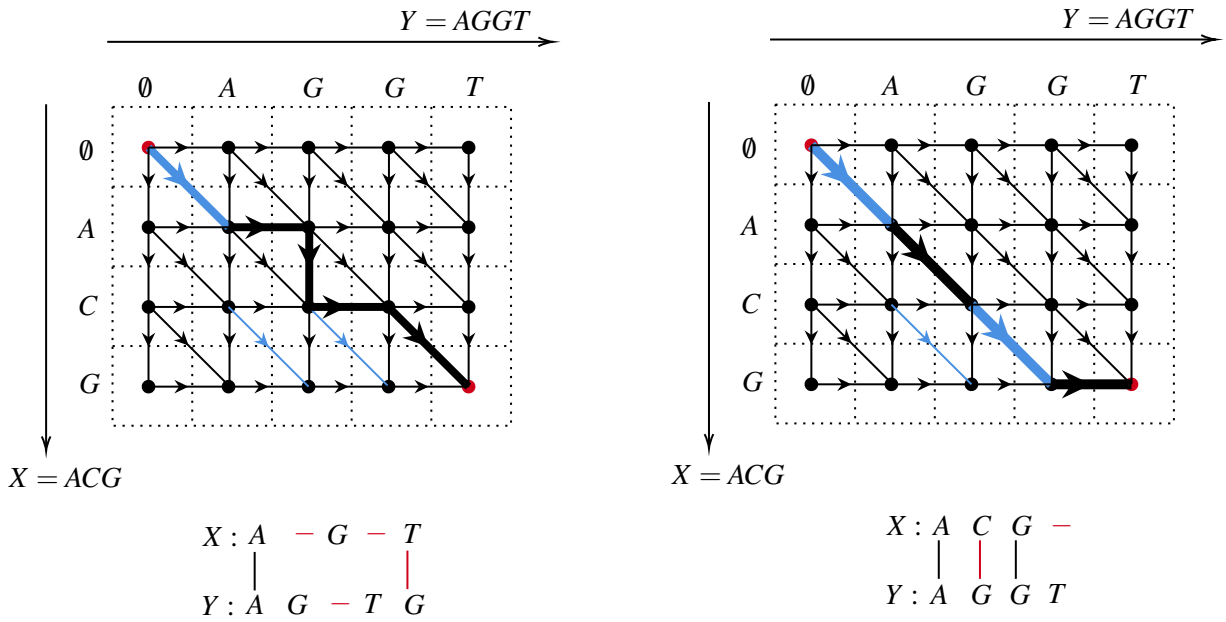



Figure 7: Two examples showing the one-to-one correspondence between paths in the graph and alignments.

from $v_{00}$ to $v_{|X|,|Y|}$. As the graph is a DAG, we can use the linear time algorithm to find the shortest path from $v_{00}$. The graph contains $\Theta(|X||Y|)$ edges and vertices. Hence the running time is $\Theta(|X||Y|)$, the same as the DP algorithm.

# Lecture 22 RNA Secondary Structure

A RNA molecule is a chain of nucleotides drawn from $\{A,C,G,U\}$. Unlike DNA that is double-stranded, in which *A-T* and *G-C* are paired with hydrogen bonds across the two strands, RNA is single-strand. To stay at a stable conformation, RNA molecule tends to *fold* by forming nucleotide-pairs within its single strand. Such self-pairing structure is called the secondary secondary structure of RNA.

We can observe some properties in RNA secondary structure. See textbook Algorithm Design [KT], page 273, for an example. First, $A$ can only be paired with $U$ and $G$ can only be paired with $C$. Second, a nucleotide can only be paired with a single nucleotide. Third, the resulting pairing must be non-crossing (also called nested). Formally, if $X[i]$ is paired with $X[j]$, $i < j$, then nucleotides within $X[i+1 \cdots j-1]$ can only be paired within themselves, i.e., it is not possible to have a pair $X[k]$ with $X[l]$ where $i < k < j$ but $l < i$ or $l > j$. If a pairing (a set of pairs) satisfies these 3 constraints, we say it is a valid pairing. See Figure 1.

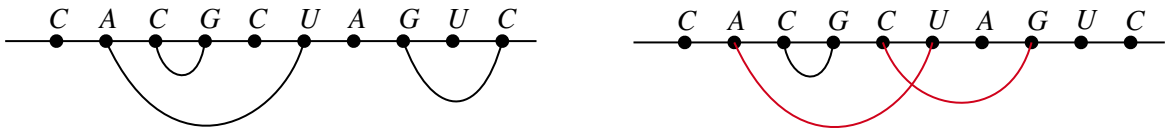



Figure 1: Left: a valid pairing; Right: a non-valid pairing as the two red pairs cross.

In general, the more pairs a RNA molecule forms, the more stable it will be. Therefore, "maximizing pairs" is a reasonable objective function in formulating RNA secondary structure problem into an optimization problem. Formally, given a string $X$ over alphabet $\Sigma = \{A,C,G,U\}$, we seek a valid pairing of $X$ such that the number of pairs is maximized.

Note: RNA folding is a *science* problem. More factors are involved and related in determining its secondary structure, and above description only considers a subset of these. Even these 3 constraints we described are not always the case (exceptions always exist in Biology). The above optimization problem is a *model* of RNA folding. It's not perfect. In fact, "all models are wrong". "Right" or "wrong" is not the proper criterion in evaluating a model. As long as a model can explain existing observations, and can predict verifiable outcomes, it is a *useful* model. After it is formualted as a mathematical problem, "right" or "wrong" criterion applies: we will design a *correct* algorithm to solve this formulated problem, and prove that it always returns the optimal solution.

## A Dynamic Programming Algorithm

We again want to design a dynamic programming algorithm. We first check if it satisfies certain optimal substructure. See Figure 2. Suppose that we know the optimal pairing of $X$, then its (nested) substructure must be the optimal pairing of the corresponding substring of $X$. Formally, let $P$ be an optimal pairing of $X[1 \cdots n]$. If $X[n]$ is not paired in $P$, then we have that the substructure of $P$ on $X[1 \cdots n-1]$ is an optimal pairing of $X[1 \cdots n-1]$. (Proof: suppose there exists a better pairing of $X[1 \cdots n-1]$, then we can concatenate it with the unpaired $X[n]$ to get a better pairing of $X$.) If $X[n]$ is paired in $P$, say, with $X[k]$, then we have that the substructure of $P$ on $X[1 \cdots k-1]$ is an optimal pairing of $X[1 \cdots k-1]$, and the substrcture of $P$ on $X[k+1 \cdots n-1]$ is an optimal pairing of $X[k+1 \cdots n-1]$. (Proof: suppose there exists a better pairing of $X[1 \cdots k-1]$, then we can use it to replace the coressponding portion of $P$ to get a better pairing of $X$. The same for $X[k+1 \cdots n-1]$.) An inductive statement leads to that smaller nested substructure is optimal w.r.t. the corresponding substring too (for example, *ACGCU*, *CGC*, etc, in Figure 2).

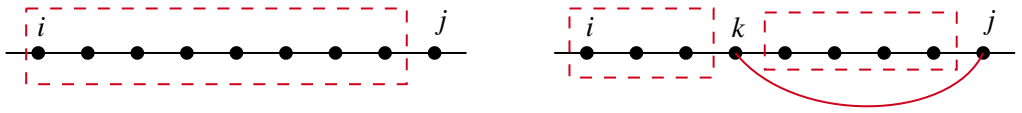

Figure 2: Left: an optimal pairing; Middle: an optimal pairing of sustring *ACGCUA*; Right: an optimal pairing of substring *GUC*.

So, this problem indeed satisfies optimal substructure property. It also suggests defining subproblems *overall substrings* (or over intervals) of $X$. Formally, we define $P(i,j)$ as the number of pairs in the optimal pairing of $X[i\cdots j]$. Clearly, the number of pairs in the optimal pairing of $X$ is $P(1,n)$ where $n=|X|$.

We now develop a recursion, which again is the reverse process of reasoning the optimal substructure but with necessary enumerations. See Figure 3 Specifically, to calculate $P(i,j)$, the optimal pairing over substring $X[i\cdots j]$, we enumerate all possibilities of the last step, i.e., how $X[j]$ is paired. The first case is that $X[j]$ is not paired, and in this case $P(i,j)=P(i,j-1)$. The second case is that $X[j]$ is paired. With whom? We don't know so we enumerate all possible partner of $X[j]$, which must be one of the $X[k]$ where $i\le k<j$ and that $X[j]$ and $X[k]$ can be paired (i.e., *A-U* or *G-C*). Once we know that $X[j]$ is paired with $X[k]$, then we have two subproblems over substrings $X[i\cdots k-1]$ and $X[k+1\cdots j-1]$. Based on above optimal substructure, these two subproblems must be optimal in order to get the optimal pairing of $X[i\cdots j]$. Therefore, in this case we have $P(i,j)=\max_{i\le k<j, X[j]\sim X[k]}(1+P(i,k-1)+P(k+1,j-1))$, where the extra 1 represents the pair of $(X[k],X[j])$ and I use $X[j]\sim X[k]$ to represent that $X[j]$ and $X[k]$ can be paired.

Combined, we have the following recursion.

$$P(i,j)=\max\begin{cases} P(i,j-1) \\ \max_{i\le k<j, X[j]\sim X[k]}(1+P(i,k-1)+P(k+1,j-1)) \end{cases}$$




case 1: X[j] is not paired

case 2: X[j] is paired with X[k]

Figure 3: Two cases in developing the recursion.

We now complete the algorithm by filling the DP table. Note that for this problem we *cannot* fill the table row-by-row with increasing row-index. See Figure 4. This is because, the the calculation of $P(i,j)$ relies on subproblems $P(k+1,j-1)$ where $k\ge i$. A feasible way is to fill the table following the diagnals. This is essentially to solve subproblems in increasing order of the length of the substrings, i.e., first solving substrings of length 1 (i.e., $(X[i],X[i])$, $1\le i\le n$), then solving substrings of length 2 (i.e., $(X[i],X[i+1])$, $1\le i\le n-1$), then solving substrings of length 3 (i.e., $(X[i],X[i+2])$, $1\le i\le n-2$), and so on.

The pseudo-code is given below. To faciliate applying the recursion, we initialize $P(i,i-1)=0$, $1\le i\le n$. We define variable $d=j-i+1$ to represent the length of substring and use it to index the outer loop; in this way, we are examining substrings with increasing length. We use variable $i$ to locate the starting position of each substring. Once $d$ and $i$ are fixed, the ending position of the substring $j$ will be $i+d-1$. The running time of this algorithm is $O(n^3)$, as the calculation of each entry takes $O(n)$ time.

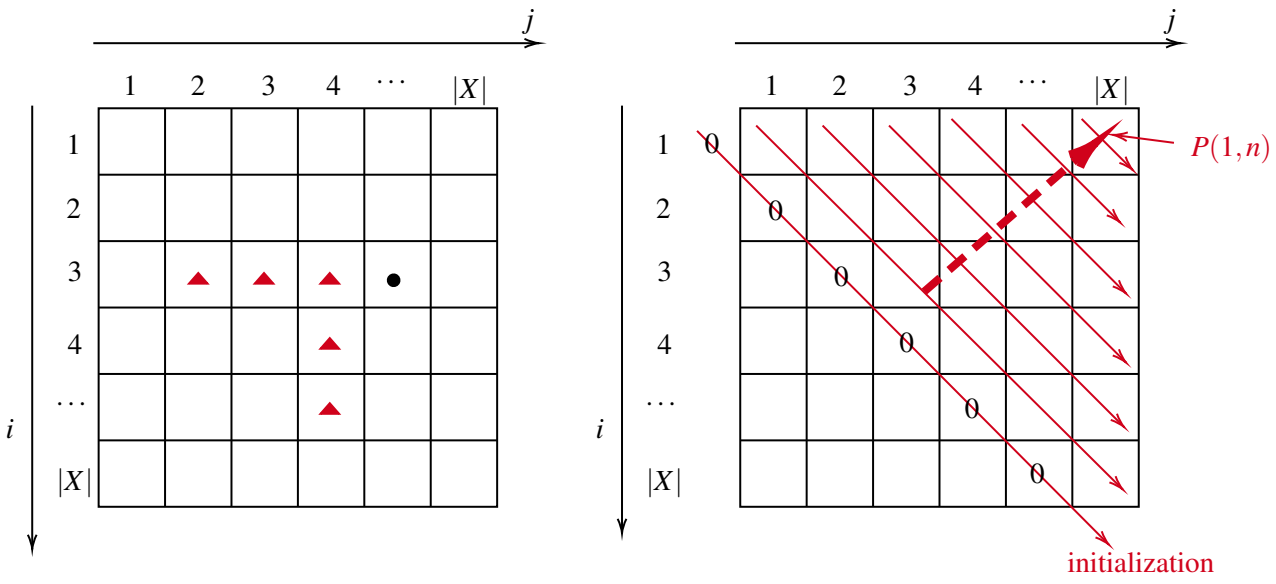



Figure 4: Left: the DP table showing the entries (in red triangles) needed by the current entry (in black dot). Right: the order to fill the table.

Algorithm rna-folding (string $X$ of length $n$)
  $P(i,i-1)=0$, for any $1 \le i \le n$
  for $d = 1 \to n$
    for $i = 1 \to n-d+1$
      $j = i+d-1$;
      $P(i,j) = P(i,j-1)$;
      for $k = i \to j-1$
        if ($X[k]$ and $X[j]$ can be paired, and $P(i,j) < P(i,k-1)+P(k+1,j-1)+1$);
          $P(i,j) = P(i,k-1)+P(k+1,j-1)+1$;
        end if;
      end for;
    end for;
  end for;
  report: $P(1,n)$;
end algorithm;

# Lecture 23 Shortest Path Problem Revisited

Let's revisit the shortest path problem with possibly negative edge length. Formally, given graph $G = (V, E)$ with edge length $l(e)$ for any $e \in E$ (note that it could be that $l(e) < 0$) and source vertex $v \in V$, to calculate the shortest path from $s$ to every $v \in V$. Here we assume that $G$ does not contain negative cycle (so the shortest path exists for every vertex that is reachable from $s$).

We first define subproblems: define $dist(k, v)$ as the length of the shortest path from $s$ to $v$ using at most $k$ edges; the "subproblem" is then to calculate $dist(k, v)$. These subproblems correspond to partition the "search space" in this way: define $X(v)$ as the set of all paths from $s$ to $v$ in $G$. Define $X(k, v)$ as the set of all paths from $s$ to $v$ using at most $k$ edges. Therefore, $dist(k, v)$ is the length of the optimal path in $X(k, v)$.

Here's connection of these subproblems with the original problem.

**Fact 1.** If $G$ does not contain negative cycle, then $dist(|V| - 1, v) = distance(s, v)$ for every $v \in V$.

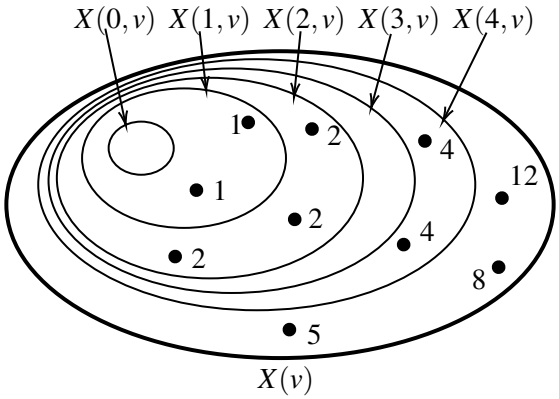



Figure 1: The illustration of partitioning the search space $X(v)$ into sub-spaces. Each point represents a path from $s$ to $v$ and the number next to it represents the number of edges in this path.

We now develop a recursion to calculate $dist(k, v)$ from $\{dist(i, u) \mid 0 \le i < k, u \in V\}$. We consider two cases. First, the shortest path from $s$ to $v$ using at most $k$ edges actually has $k - 1$ or less edges; in this case we have $dist(k, v) = dist(k - 1, v)$. Second, the shortest path from $s$ to $v$ has $k$ edges. In this case, we consider all possibilities of the last edge in the optimal path. (Enumerating all possible cases of the last step is a commonly-used approach in designing dynamic programming algorithms.) Such edge must be one of the in-edges of $v$. Once the last edge of the optimal path, say $(u, v)$, is fixed, then we know the remaining part of the optimal path must be the optimal path from $s$ to $u$ with at most $(k - 1)$ edges. With this analysis, if it is the second case, then $dist(k, v) = \min_{(u,v)\in E}(dist(k - 1, u) + l(u, v)$. Combined, the recursion is

$$dist(k, v) = \min\{dist(k - 1, v), \min_{(u,v)\in E}(dist(k - 1, u) + l(u, v)\}.$$

(Think: would this recursion $dist(k, v) = \min_{(u,v)\in E}(dist(k - 1, u) + l(u, v))$ work? Why?)

With above recursion available, the algorithm is rather straightforward. The algorithm is simply to fill the dynamic programming table. In this case, the table is $|V| \times |V|$: each row is indexed by $k$, $0 \le k \le |V| - 1$, and each column is indexed by $v \in V$.

A typical dynamic programming algorithm can be organized into 3 steps. The *initialization step*, in this case, fills the first row of the table: $dist(0, s) = 0$ and $dist(0, v) = \infty$ if $v \ne s$. The *iteration step* fills all other rows of the table using above recursion. The *termination step* answers the original question, in this case,

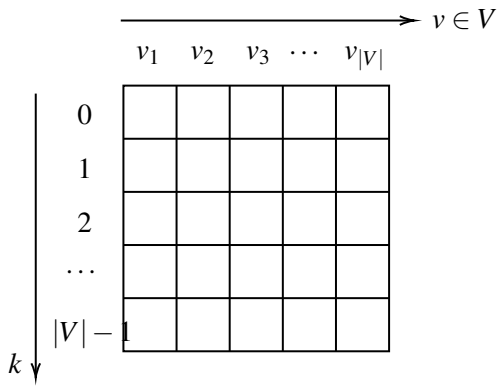
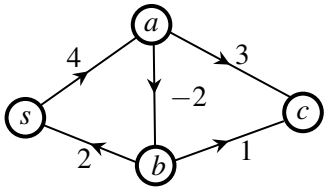



Figure 2: Illustration of the dynamic programming table for shortest-path problem.

reports that the last row gives the distance for every vertex. The pseudo-code is given below. Try run this algorithm (fill the dynamic programming table) on the example below.

```
Algorithm DP-shortest-path (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    init a 2D array dist of size |V| × |V|;
    dist[0,s] = 0; dist[0,v] = ∞ for any v ≠ s;
    for k = 1 → |V| − 1;
        for v ∈ V;
            dist(k,v) = dist(k−1,v);
            for each (u,v) ∈ E;
                if (dist(k,v) > dist(k−1,u) + l(u,v))
                    dist(k,v) = dist(k−1,u) + l(u,v);
                end if;
            end for;
        end for;
    end for;
    dist[|V| − 1,v] gives distance(s,v), for any v ∈ V;
end algorithm;
```




| | $s$ | $a$ | $b$ | $c$ |
|---|---|---|---|---|
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | 4 | $\infty$ | $\infty$ |
| 2 | 0 | 4 | 2 | 7 |
| 3 | 0 | 4 | 2 | 3 |

Figure 3: An example of running above DP algorithm.

The second for-loop and the the third for-loop together take $O(|E|)$ time, as it examines, for each $v \in V$, the in-edges of $v$. Hence, the running time of this algorithm is $O(|V| \cdot |E|)$.

## Recursion Revisited

In the core recursion of the DP algorithm, $dist(k,\cdot)$ only depends on $dist(k-1,\cdot)$. In other words, the calculation of next row of the DP table only relies on the value of the previous row. This property has the following two applications.

**Early Termination.** In any run of the DP algorithm, if we observe that the $k$-th row and $(k+1)$-th row are identical, then the algorithm can be terminated (rather than keep running it until the $(|V|-1)$-th row), as we know that any of the future row will be identical to these two rows.

**Space Complexity.** It seems that the DP algorithm requires $|V|^2$ space, as the DP table is $|V| \times |V|$. In fact, this can be improved to $2|V|$, as we only need to maintain two rows, the current row and the previous row; the completion of the current row uses the values in the previous row, and after the current row has been completed, copy it to the previous row.

## Characterization of Negative Cycle

We focus on this problem: given graph $G = (V,E)$ with edge length $l(e)$, $e \in E$, and a source vertex $s$, to decide if there exists a negative cycle in $G$ that is reachable from $s$. Before we design algorithm for it, we first derive an equivalent characterization.

Recall that $dist(k,v)$ is defined as the length of the shortest path from $s$ to $v$ using at most $k$ edges. We emphasize that, this definition is still valid in the presence of negative cycles. Comparing with $distance(s,v)$, which may not be well-defined as the shortest path may use unlimited number of edges, the definition of $dist(k,v)$ limits the number of edges (i.e., $k$) in the path. Hence, $dist(k,v)$ is well-defined for any $k \geq 0$ and any $v \in V$. See an example below.

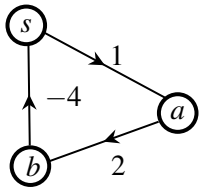



Figure 4: Example showing that $dist(k,v)$ is well-defined in the presence of negative cycle. In this example, $dist(1,a) = 1$, $dist(301,a) = -99$, $dist(3001,a) = -999$.

**Fact 2.** Let $C$ be a negative cycle reachable from $s$; let $v \in C$. Then we have $\lim_{k\to\infty} dist(k,v) = -\infty$.

*Proof.* This is because, one can loop within $C$ to get shorter and shorter path from $s$ to $v$. □

## DP Algorithm to Detect Negative Cycle

We will extend above DP algorithm to do it. Recall that the DP consists of initialization, iteration, and termination steps. We emphasize that, in the presence of negative cycles, the initialization and iteration steps are still correct (simply because the proof of their correctness doesn't assume that $G$ does not contain negative cycle), i.e., we can still use them to calculate $dist(k,v)$, for *any* $k \geq 0$, even if $G$ contains negative cycle. It's only that the termination step won't work in the presence of neagive cycle, as $dist(|V|-1,v) = distance(s,v)$ needs the condition that $G$ does not contain negative cycle. Note too that the early-termination property is also correct in the presence of negative cycles.

Here comes the algorithm to decide negative cycle reachable from $s$: it simply calculates one more row in the DP table and check if the last two rows are identical.

```
Algorithm detect-negagive-cycle (G = (V,E), l(e) for any e ∈ E, s ∈ V)
    fill one more row in the DP table, i.e., calculate dist(|V|,v) for any v ∈ V;
    if (dist(|V|,v) = dist(|V|−1,v) for every v ∈ V)
        report: "G does not contain negative cycle reachable from s";
    else
        report: "G contains negative cycle reachable from s";
    end if;
end algorithm;
```

We now prove the correctness of the above algorithm. The proof is a combination of the facts we have seen, namely Fact 1 in Lecture 19, the "early-termination" property, and Fact 1 in this lecture.

*Proof.* We first prove that, if $G$ does not contain negaive cycle reachable from $s$, then $dist(|V|,v) = dist(|V|-1,v)$ for every $v \in V$. Following Fact 1 in Lecture 19, we have $dist(|V|-1,v) = distance(s,v)$. Note that $dist(|V|,v) \leq dist(|V|-1,v)$, as $dist(|V|,v)$ is the optimal solution of a larger sub-space than that of $dist(|V|-1,v)$. Note too that $dist(|V|,v) \geq distance(s,v)$, as $distance(s,v)$ is the optimal solution of the entire search space (and $distance(s,v)$ is well-defined without negative cycle). Combined, $dist(|V|,v) = dist(|V|-1,v)$ for every $v \in V$.

We then prove that, if $G$ contains negaive cycle reachable from $s$, then there exist $v \in V$ such that $dist(|V|,v) \neq dist(|V|-1,v)$. Suppose conversely that $dist(|V|,v) = dist(|V|-1,v)$ for every $v \in V$. Then following the early-termination property, any future row will be the same as $dist(|V|-1,v)$, i.e., $\lim_{k\to\infty} dist(k,v) = dist(|V|-1,v)$, for every $v \in V$. This contradicts to the Fact 1 in this lecture. □

## Floyd-Warshall Algorithm

We consider the *all-pairs shortest path problem*: given graph $G = (V,E)$ with edge length $l(e)$ for any $e \in E$, to compute $distance(u,v)$ for any $u,v \in V$. A straitforward algorithm is to run any algorithm for single-source shortest path problem $|V|$ times, in each of which starting from a different vertex as source. The running time will be $O(|V|^2|E|)$.

We now introduce Floyd-Warshall algorithm, which takes $O(|V|^3)$ time. Instead of limiting the number of edges in the shortest paths as used in the DP algorithms for single-source shortest path problem, in Floyd-Warshall algorithm, we limit the *largest index of the intermediate vertices* in the shortest paths. Specifically, we rename $V = \{v_1, v_2, \cdots, v_n\}$, where $n = |V|$. We define $dist(i,j,k)$ as the length of the shortest path from $v_i$ to $v_j$ such that the intermediate vertices on the path is from $\{v_1, v_2, \cdots, v_k\}$. In other words, among all paths from $v_i$ to $v_j$ with intermediate vertices being from $\{v_1, v_2, \cdots, v_k\}$, $dist(i,j,k)$ gives the length of the shortest one. Clearly, we have that $distance(v_i, v_j) = dist(i,j,n)$.

Now let's develop a recursion. Let $p$ be the shortest path from $v_i$ to $v_j$ with intermediate vertices being from $\{v_1, v_2, \cdots, v_k\}$. There are two possibilities: $p$ does not use $v_k$ and $p$ uses $v_k$. If it is the first case, we know that $dist(i,j,k) = dist(i,j,k-1)$. If it is the second case, $p$ is partitioned into two paths, one from $v_i$ to $v_k$ and the other is from $v_k$ to $v_j$, and in either path the intermediate vertices must be from $\{v_1, v_2, \cdots, v_{k-1}\}$. Therefore, in this case, $dist(i,j,k) = dist(i,k,k-1) + dist(k,j,k-1)$. Combined, we have the following recursion:

$$dist(i,j,k) = \min\{dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1)\}.$$

The complete algorithm fills the table shown above. The pseudo-code is given below.

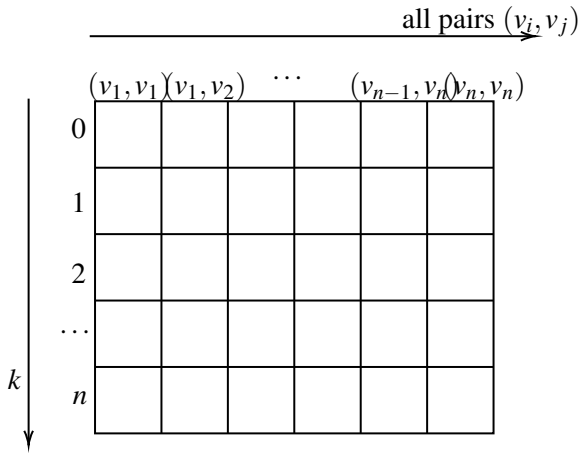



Figure 5: The DP table of the Floyd-Warshall algorithm.

Algorithm Floyd-Warshall ($G = (V,E)$, $l(e)$ for any $e \in E$
    init a 3D array *dist* of size $|V| \times |V| \times |V|$, initialized as $\infty$;
    $dist[i,i,0] = 0$, $1 \leq i \leq n$;
    $dist[i,j,0] = l(v_i, v_j)$, for any $(v_i, v_j) \in E$;
    for $k = 1 \rightarrow n$;
        for $i = 1 \rightarrow n$;
            for $j = 1 \rightarrow n$;
                $dist(i,j,k) = \min\{dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1)\}$;
            end for;
        end for;
    end for;
    report: $dist[i,j,n]$ gives $distance(v_i, v_j)$, for any pair $v_i, v_j \in V$;
end algorithm;

# Lecture 24 Knapsack Problem

Given a knapsack with total capacity of $W$, and $n$ items, where item-$i$ has weight $w_i$ and value $v_i$, the *knapsack problem* is to select a bunch of items, represented as a vector $(x_1, x_2, \cdots, x_n)$, where $x_i$ gives how many item-$i$ are picked, such that the total weight of these items is at most $W$, i.e., $\sum_{1 \le i \le n} x_i w_i \le W$, and that the total value of them, i.e., $\sum_{1 \le i \le n} x_i v_i$, is maximized.

There are two variants of above knapsack problem. The first variant (with repetition) allows each (type of) item can be selected unlimitted number of times, i.e., $x_i \in \{0, 1, 2, \cdots\}$, $1 \le i \le n$. The second variant (without repetition) requires that each item can be selected at most once, i.e., $x_i \in \{0, 1\}$, $1 \le i \le n$.

Consider an instance with total capacity $W = 7$ and 4 items with weights being $(5, 2, 1, 3)$ and values being $(8, 3, 2, 7)$. For the variant 1, the optimal solution is $(0, 0, 1, 2)$, i.e., item 3 is picked once and item 4 is picked twice, which gives a total value of 16 and a total weight of 7. For the variant 2, the optimal solution is $(0, 1, 1, 2)$, i.e., item 2, 3 and 4 are picked (once), which gives a total value of 12 and total weight of 6.

## An Exercise

Since we want to maximize the total value given the capacity limit, an intuitive strategy is therefore to pick "the most valuable item per unit". This leads to a greedy algorithm. Formally, we sort all items w.r.t. $v_i/w_i$ in descending order, and then pick items (greedily) following this order: if the current item can be fit into the knapsack, we pick it, reduce the total capacity by its weight, and then either stay at this item (for variant 1) or move to next item (for variant 2); if the weight of the current item exceeds the remaining capacity of the knapsack, move to the next item. This algorithm is good heuristic and unfortunately may not always find the optimal solution.

1. Design an instance for variant 1 such that this algorithm fails to find the optimal solution.
2. Design an instance for variant 2 such that this algorithm fails to find the optimal solution.

## Algorithm for Variant 1

We now design an optimal, dynamic programming algorithm for variant 1. Again, let's first check if it satisfies optimal substructure property. Let $p = (x_1, x_2, \cdots, x_n)$ be the optimal solution. Assume that $x_k \ge 1$. Let $q$ be the solution obtained from $p$ by reducing $x_k$ by 1, i.e., $q = (x_1, x_2, \cdots, x_{k-1}, x_k - 1, x_{k+1}, \cdots, x_n)$. Then $q$ is the optimal solution when the total capacity is $W - w_k$. (Proof: suppose $q$ is not the optimal solution with total capacity being $W - w_k$, i.e., one can find a solution achieves higher value than $q$ and does not exceed capacity limit of $W - w_k$. Then adding item-$k$ to this solution yields a better solution than $p$, contradicting to the assumption of $p$.)

According to this optimal substructure property, naturally, we define $F(a)$ as the total achievable value with capacity limit being $a$. Clearly $F(W)$ answers the original question. We now develop the recursion. In order to calculate $F(a)$, following above optimal substructure property, we need to find an item in the optimal solution so as to reduce it to a smaller subproblem. Which item is in the optimal solution? We don't know. So we enumerate. We consider every single item that fits (i.e., weight is less or equal to $a$). Once we know, say item-$k$, is in the optimal solution, the remaining part of the optimal solution can be obtained by calculating the optimal solution with $a - w_k$ being the total capacity. Hence, the recursion is

$$F(a) = \max_{1 \le k \le n, w_k \le a} (F(a - w_k) + v_k).$$

The algorithm simply fills the array of size $(W + 1)$, i.e., $F(0), F(1), \cdots F(W)$, with this recursion. The running time is therefore $O(nW)$, as solving a single subproblem takes $O(n)$ time.

Note that this algorithm does not run in polynomial-time. This is because, the *input size* of numerical value $W$ is $\log W$. By rewriting the running time as a function of input-size, i.e., $O(nW) = O(n2^{\log W})$, we can see the running time is exponential. (Note: if the running time is a polynomial function of *numerail values*, we call it *pseudo-polynomial time* algorithm. And this algorithm is such an example.)

## Algorithm for Variant 2

Note that the optimal substructure property stated above for variant 1 does not hold for variant 2 anymore. This is because in variant 2 each item can be picked at most once. Therefore, another parameter should be introduced to "memorize" which items are available in the subproblem. A convenient and efficient way is to limit the largest index of available items. Here is the optimal substructure property for variant 2: let $p = (x_1, x_2, \cdots, x_n)$ be the optimal solution (recall that each $x_i \in \{0,1\}$). Let $k$ be the largest index with $x_k = 1$. Let $q$ be the solution obtained from $p$ by reducing $x_k$ by 1, i.e., $q = (x_1, x_2, \cdots, x_{k-1}, x_k - 1, x_{k+1}, \cdots, x_n)$. Then $q$ is the optimal solution when the total capacity is $W - w_k$ and only first $(k-1)$ items are available. (Proof: suppose $q$ is not the optimal solution, i.e., one can find a set of items from the first $(k-1)$ items that achieves higher value (than $q$) and does not exceed capacity limit of $W - w_k$. Then adding item-$k$ to this solution yields a better and feasible solution than $p$, contradicting to the assumption of $p$.)

Following above optimal substructure property, we define $F(k,a)$ as the total achievable value using the first $k$ items with capacity limit being $a$. Clearly $F(n,W)$ answers the original question. We now develop the recursion. Consider the last step, i.e., whether item-$k$ is included in the optimal solution. In the case that the optimal solution doesn't include item-$k$ we have $F(k,a) = F(k-1,a)$. In the case that the optimal solution includes item-$k$ we have $F(k,a) = F(k-1, a-w_k) + v_k$. Note that in order to be able to include item-$k$, it is required that $w_k \leq a$. Hence, the recursion is

$$F(k,a) = \begin{cases} \max \begin{cases} F(k-1,a-w_k) + v_k & \\ F(k-1,a) & \end{cases} & \text{if } w_k \leq a \\ F(k-1,a) & \text{if } w_k > a \end{cases}$$

The algorithm fills out the DP table of size $n(W+1)$ with above recursion. The running time is therefore $O(nW)$, as solving a single subproblem takes $O(1)$ time. Again this is a *pseudo-polynomial time* algorithm.

# Lecture 25 Why DP Table? Why not Recursive Programming?

In all the DP algorithms we've designed, we always use a table to store the optimal solution of subproblems; when solving larger subproblems, we simply query the table and fetch the optimal solutions as needed. This leads to a bottom-up strategy, i.e., we start from solving smaller subproblems as they will be needed by larger ones. Is this strategy always necessary? Can we directly solve the original problem with recursive programming?

Let's try. Consider the edit distance problem (Lecture 26). The DP algorithm we designed there uses the recursion below.

$$F(i,j) = \min \begin{cases} F(i-1,j-1) + \delta(X[i] \neq Y[j]) \\ F(i-1,j)+1 \\ F(i,j-1)+1 \end{cases}$$

We write a recursive procedure: define $F(i,j)$ returns the edit distance between $X[1\cdots i]$ and $Y[1\cdots j]$.

function $F(i,j)$
  if $i = 0$: return $j$;
  if $j = 0$: return $i$;
  $z_1 = F(i-1,j-1) + \delta(X[i] \neq Y[j])$;
  $z_2 = F(i-1,j)+1$;
  $z_3 = F(i,j-1)+1$;
  return $\min\{z_1,z_2,z_3\}$;
end function;

We then call $F(m,n)$, where $m = |X|$ and $n = |Y|$. This is a correct algorithm. How about its running time? We define $T(m,n)$ as the running time of $F(m,n)$. We have $T(m,n) = T(m-1,n-1) + T(m-1,n) + T(m,n-1) + 1$. To *approximate* $T(m,n)$, assume that $m = n$. Notice also that $T(m-1,n) \geq T(m-1,n-1)$ and $T(m,n-1) \geq T(m-1,n-1)$. We can write $T(n,n) \geq 3T(n-1,n-1)+1$. Hence, $T(n,n) \geq \Theta(3^n)$. So, above algorithm runs in exponential-time! This suggests that recursive procedure is a bad strategy.

Note that there are $\Theta(mn)$ distinct subproblems, i.e., $F(i,j)$, $0 \leq i \leq m$, $0 \leq j \leq n$. The reason why above algorithm runs in exponential time is that it keeps solving the *same* subproblems. See Figure 1. This again suggests the necessity of maintaining a table to store the optimal solutions so as to avoid repeating resolving the same ones. An alternative way to understand it is that, subproblems will be used multiple times by

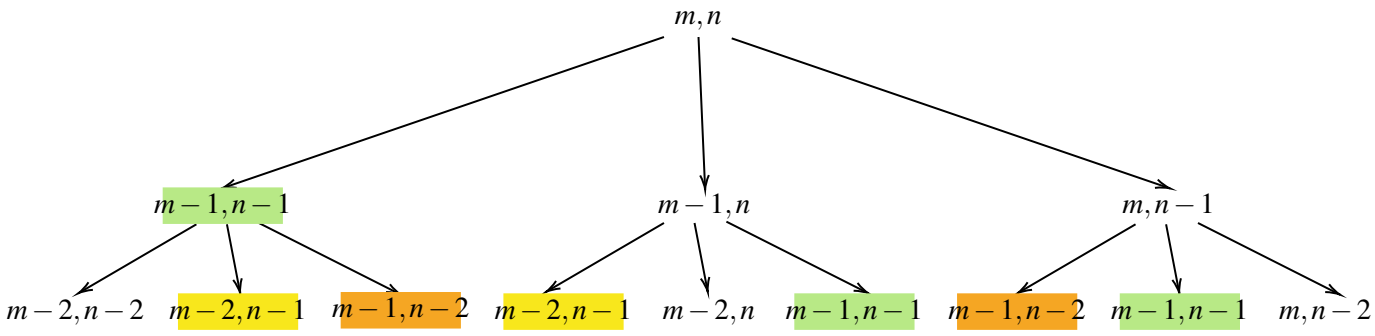



Figure 1: First 3 levels of the recursive tree of above procedure. Identical recursive calls are marked with the same color.

different larger subproblems, hence it's beneficial to store them.

Both divide-and-conquer algorithms and dynamic programming algorithm rely on a recursive formula that reduces larger subproblems into smallers. Why for D&C algorithm we don't maintain a table? There are two reasons. First, in (most) D&C algorithms, subproblems don't overlap (while in DP algorithms they do, as showed in Figure 1), or in other words, each subproblem will be used only once. Therefore there is no difference between solving recursively or by using a table. Second, in (most) D&C algorithms, the size of the subproblem is usually reduced by a *fixed ratio* (for example, $n \to n/2$), and consequently the height of the recursive tree is $\Theta(\log n)$. Hence there is *no need* to maintain a table, as solving all subproblems on the fly is still efficient. On the contrary, in (most) DP algorithms, the size of the subproblem is usually reduced by a *constant* (for example, $n \to n-1$), and consequently the height of the recursive tree is $\Theta(n)$; solving all subproblems on the fly will require exponential-time, as we just showed.

# Lecture 26 Minimum Spanning Tree

## Tree and Spanning Tree

A tree is an undirected graph that is connected and acyclic. ("connected" here means the graph consists of a single connected component.) A tree $T = (V, E)$ always satisfies that $|E| = |V| - 1$. In fact, any two of these three properties (i.e., connected, acyclic, and $|E| = |V| - 1$) implies the other one, formally stated below. Play with some examples to fully understand these properties.

**Fact 1.** Let $T = (V, E)$ be an undirected graph. The following statements are equivalent.

1. $T$ is a tree.
2. $T$ is connected and acyclic.
3. $|E| = |V| - 1$ and $T$ is acyclic.
4. $|E| = |V| - 1$ and $T$ is connected.
5. There exists an unique path between every pair of vertices of $T$.

Let $G = (V, E)$ be an undirected graph. We say tree $T = (V_1, E_1)$ is a *spanning tree* of $G$ if $V_1 = V$ and $E_1 \subset E$. As $T$ is a tree, clearly we have $|E_1| = |V_1| - 1 = |V| - 1$. Note that a spanning tree is always w.r.t. a graph, i.e., we always say a spanning tree of a graph. Note too, that for an undirected graph $G$, its spanning tree may not be unqiue. See Figure 1 for an example. (Think: when an undirected graph has a unique spanning tree?)

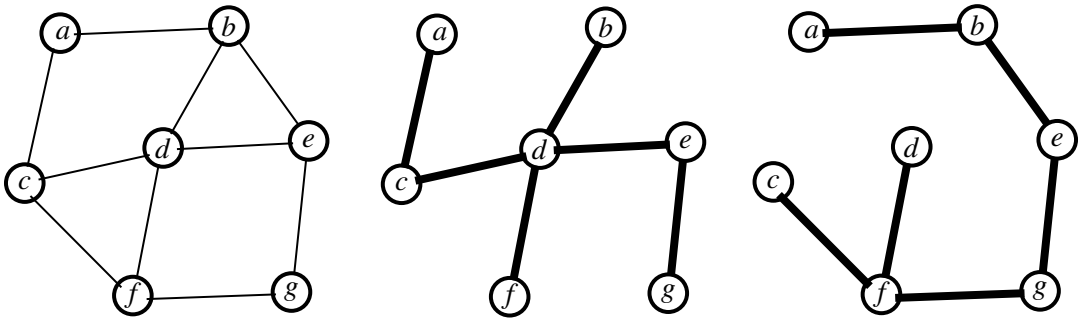



Figure 1: Two spanning trees (middle and right) of an undirected graph (left).

By above definition, a spanning tree of $G = (V, E)$ is a tree that uses $(|V| - 1)$ edges of $G$ that connects all vertices of $G$. Following Fact 1, we can construct a spanning tree of $G$ in the following two ways:

1. pick $(|V| - 1)$ edges of $G$ that are acyclic;
2. pick $(|V| - 1)$ edges of $G$ that connects all vertices of $G$;

These two approaches are equivalent but leads to different algorithms in constructing the minimum spanning tree (see below).

## Minimum Spanning Tree (MST)

Now we introduce the minimum spanning tree problem. Given an directed graph $G = (V, E)$ with *edge weight* $w(e)$ for any $e \in E$, we seek a spanning tree $T = (V, E_1)$ of $G$ such that the total weight of $T$, defined

as $\sum_{e\in E_1} w(e)$, is minimized. Such tree $T$ is also called a minimum spanning tree (MST) of $G$. MST problem has been widely used in in real-world applications. Essentially, this is because MST is the most efficient way to connect all vertices: one can use edge weight to model the cost of connecting two vertices, and a MST gives a way to connect all vertices with minimized total costs.

We now design algorithms to solve the MST problem. We will use a new algorithm-design technique: *greedy*. In general, a greedy algorithm iteratively constructs a solution, and in each iteration, chooses the optimal solution based on the *current* state. In other words, a greedy algorithm consists of a series of *local-optimal* decisions. Whether or not a greedy algorithm results in a *global* optimal solution needs a proof.

Recall that a spanning tree of $G = (V,E)$ consits of $(|V|-1)$ edges. As an MST seeks a spanning tree with minimized total weights, we are therefore motivated to pick edges with small weights when constructing the spanning tree—this is a greedy strategy. Recall we have two different ways to construct a spanning tree, i.e., either pick $(|V|-1)$ edges that are acyclic, or pick $(|V|-1)$ edges that connects all vertices. These two approaches, together with the greedy strategy, lead to two greedy algorithms, the Kruskal's algorithm and the Prim's algorithm.

## Kruskal's Algorithm

Kruskal's algorithm can be summarized as: *iteratively pick* $(|V|-1)$ *edges, and in each iteration, pick the smallest edge that doesn't produce a cycle*. The pseudo-code is given below, where we use $E_1$ to store the edges picked. See an example in Figure 2.

```
Algorithm Kruskal's (graph G = (V,E))
    sort all edges in increasing order w.r.t. weights;
    init E_1 = ∅;
    for e ∈ E in above order
        if (E_1 ∪ {e} doesn't form a cycle)
            E_1 = E_1 ∪ {e};
            if (|E_1| = |V| − 1): report E_1 and exit;
        end if;
    end for;
end algorithm;
```

Note that above algorithm is a framework, as how to decide if $E_1 \cup \{e\}$ produces cycles is not specified. Later on we will introduce a new data structure, namely *disjoint set*, to efficiently do that. We also need to prove the correctness of the Kruskal's algorithm, i.e., it always finds an MST; we will do that later on.

## Prim's Algorithm

Prim's algorithm combines the greedy strategy (i.e., pick the smallest edge whenever possible) and the 2nd way to construct a spanning tree (i.e., pick $(|V|-1)$ edges that connects all vertices). To connect all vertices, we can start from an *arbitrary* vertex and iteratively make it connected to all other vertices. As we need to use $(|V|-1)$ edges to connect to all other $(|V|-1)$ vertices, therefore every edge we pick needs to be connected to a *new* vertex. Prim's algorithm can be summarized as: *iteratively pick* $(|V|-1)$ *edges, and in each iteration, pick the smallest edge that connects to a new vertex*. We again use $E_1$ to store the picked edges. We also use $S$ to store the vertices that have been connected. The pseudo-code is given below. See an example in Figure 3.

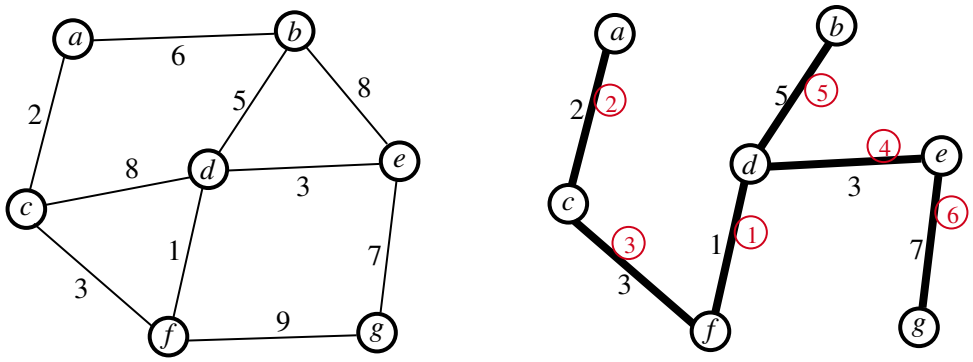



Figure 2: Running Kruskal's algorithm. The cycled numbers give the order edges are picked.

Algorithm Prim's (graph $G = (V, E)$)
  init $E_1 = \emptyset$;
  init $S$ with an arbitrarily picked vertex;
  for $k = 1 \to |V| - 1$
    pick the smallest edge $e = (u, v)$ that connects to a new vertex (i.e., $u \in S$ and $v \notin S$);
    $E_1 = E_1 \cup \{e\}$;
    $S = S \cup \{v\}$;
  end for;
end algorithm;

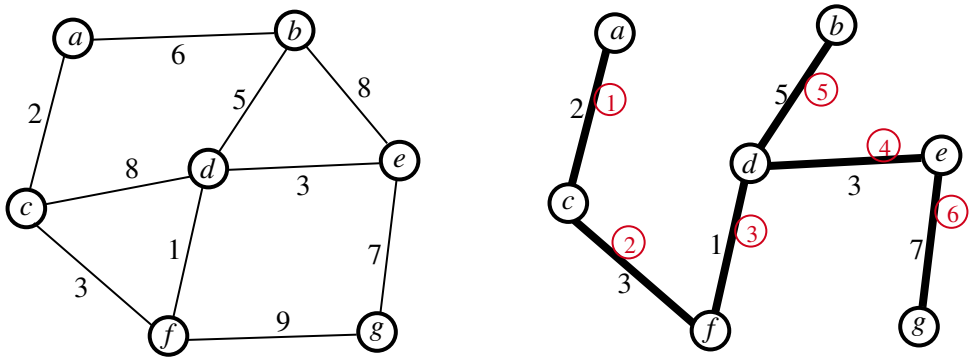



Figure 3: Running Prim's algorithm with $S$ being initialized as $\{a\}$. The cycled numbers give the order edges are picked.

Again the above algorithm is a framework, as how to select the smallest edge that connects to a new vertex is not specified. Later on we will use *priority queue* to efficiently do that. We also need to prove the correctness of the Prim's algorithm, i.e., it can always find an MST; we will do that later on.

# Lecture 27 Cut Property and Connectness of Algorithms for MST

## Cut, Cut-Edges, and Cut Property

In order to develop a key property that proves the correctness of both Kruskal's algorithm and the Prim's algorithm, we first introduce the definition of *cut* and *cut-edges*. Let $G = (V, E)$ be a graph. A *cut* $C$, denoted as $C = (S, V \setminus S)$, is a partition of $V$ (i.e., $S$ and $V \setminus S$ where $S \subset V$). Let $C = (S, V \setminus S)$ be a cut of $G$, we define the *cut-edges* w.r.t. $C$, denoted as $E(C)$, as $E(C) := \{(u,v) \in E \mid u \in S, v \in V \setminus S\}$. See examples in Figure 1. Note the difference of cut-edges when applied to undirected graphs and directed graphs.

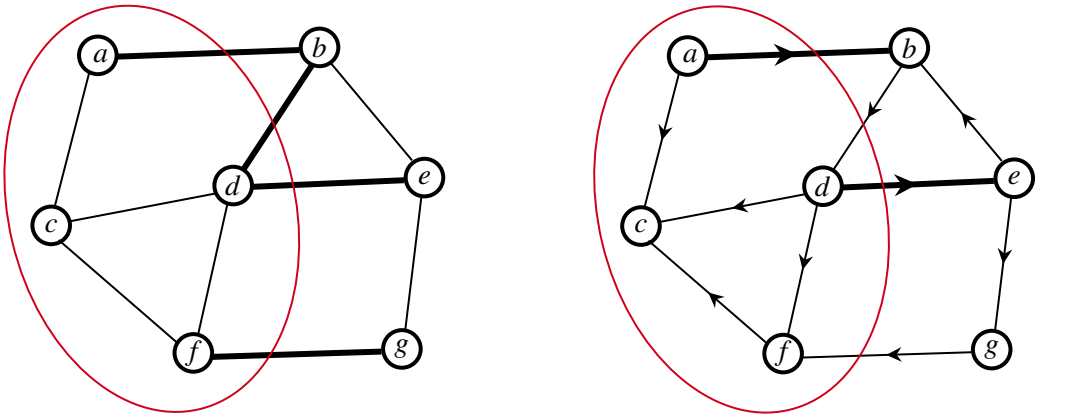



Figure 1: Cut and cut-edges on an undirected graph (left) and on a directed graph (right). Cut-edges are highlighted with bold edges.

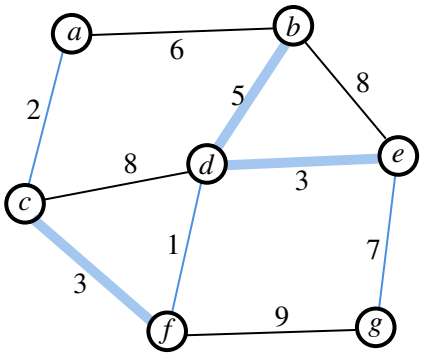
The cut-property describes how to *augment a partial MST*, formally described below. A *partial* MST is a subset of edges of a MST (see Figure 2). This property gives a way to iteratively construct a MST: one can iteratively apply it to augment a partial MST (intially empty) until it becomes a (complete) MST (i.e., its size becomes $|V| - 1$). In fact, both Kruskal's algorithm and Prim's algorithm are essentially using this property. Their correctness therefore can be proved with this property too.

**Claim 1** (Cut-Property)**.** Let $E_1$ be a partial MST of graph $G = (V, E)$. Let $C = (S, V \setminus S)$ be a cut of $G$. If $E(C) \cap E_1 = \emptyset$, then $E_1 \cup \{e^*\}$ is also a partial MST, where $e^* := \arg\min_{e \in E(C)} w(e)$.

In other words, suppose you have a partial solution $E_1$ that you know are part of a MST (i.e., there exists a MST $T^* = (V, E^*)$ such that $E_1 \subset E^*$). In order to augment $E_1$, you will need to find a cut $C = (S, V \setminus S)$ of $G$. If the cut-edges w.r.t. $C$ doesn't overlap with $E_1$ (i.e., $E(C) \cap E_1 = \emptyset$), then the smallest cut-edge, i.e., $e^* := \arg\min_{e \in E(C)} w(e)$, can be safely included.

See above example. The blue edges (including thin and thick ones) gives a MST $E^*$, and the thick ones, denoted as $E_1$, represents a partial MST. Try following cuts $C = (S, V \setminus S)$ where $S$ is specified:

1. $S = \{a\}$. $E(C) = \{(a,c),(a,b)\}$. $E(C) \cap E_1 = \emptyset$. $e^* = (a,c)$. So $E_1 \cup \{e^*\}$ is a partial MST.
2. $S = \{a,c,f\}$. $E(C) = \{(a,b),(c,d),(f,d),(f,g)\}$. $E(C) \cap E_1 = \emptyset$. $e^* = (d,f)$. So $E_1 \cup \{e^*\}$ is a partial MST.




$E^*$={(a,c),(c,f),(f,d),(d,b),(d,e),(e,g)}

$E_1$={(c,f),(d,b),(d,e)}

Figure 2: Illustrating the cut-property.

3. $S = \{a,g\}$. $E(C) = \{(a,b),(a,c),(f,g),(e,g)\}$. $E(C) \cap E_1 = \emptyset$. $e^* = (a,c)$. So $E_1 \cup \{e^*\}$ is a partial MST.

4. $S = \{a,c\}$. $E(C) = \{(a,b),(c,d),(c,f)\}$. $E(C) \cap E_1 = \{(c,f)\}$. So we can't get a larger partial MST with this cut.

Note that, a partial MST $E_1$ partitions all vertices of $G$ into disjoint connected components. (Like in above example there are 4 components: $\{a\},\{c,f\},\{b,d,e\},\{g\}$, formed by $E_1$.) Clearly, a cut $C = \{S, V \setminus S\}$ satisfies $E(C) \cap E_1 = \emptyset$ if and only if for each component either $S$ includes it entirely or excludes it entirely; in other words in order not to overlap with $E_1$, a cut just needs to avoid "breaking" any connected components formed by $E_1$.

We now prove above claim. See Figure 3. The setup is a MST $E^*$, a partial MST $E_1 \subset E^*$, and a cut $C = (S, V \setminus S)$ satisfying that $E(C) \cap E_1 = \emptyset$. And we need to prove that $E_1 \cup \{e^*\}$ is a (partial) MST too, where $e^*$ is the smallest cut-edge w.r.t. $C$.

Consider two cases. If $e^* \in E^*$, (in Figure 3, this means $e^*$ is one of $\{(a,c),(c,d),(d,f),(d,e)\}$), then obvisously in this case $E_1 \cup \{e^*\}$ is a (partial) MST, and this MST is $E^*$. This is a trivial case.

The second case is that $e^* \notin E^*$. (In Figure 3, this means $e^*$ is one of $\{(b,e),(b,h)\}$.) Assume that $e^* = (u,v)$ where $u \in S$ and $v \notin S$. (In Figure 3, w.l.o.g., we assume $e^* = (b,h)$, i.e., $u = b$ and $v = h$.) Since $E^*$ is a spanning tree, there exists a unique path $p$ from $u$ to $v$ using edges in $E^*$. (In Figure 3, path $p$ is $(b,a) \to (a,c) \to (c,d) \to (d,e) \to (e,i) \to (i,h)$.) Since $u \in S$ and $v \notin S$, path $p$ must "cross" cut $C$, i.e., we must have that edges of $p$ overlaps with cut-edges: $p \cap E(C) \neq \emptyset$. (In Figure 3, $p \cap E(C) = \{(a,c),(c,d),(d,e)\}$.) Pick an arbitrary edge $e'$ from $p \cap E(C)$ and build $E'$ by replacing $e'$ with $e^*$, i.e., $E' = E^* \setminus \{e'\} \cup \{e^*\}$. (In Figure 3, we can pick $e' = (a,c)$ and build $E' = \{(a,b),(c,d),(d,f),(d,e),(e,g),(e,i),(h,i),(b,h)\}$.) We can show that $E'$ is still a spanning tree of $G$, as $|E'| = |E^*|$ and it connects all vertices (removing $e'$ breaks all vertices into 2 components where $u$ and $v$ are separated into different components, but adding $e^* = (u,v)$ glues them together). How about the total weight of $E'$? Clearly, $w(E') = w(E^*) - w(e') + w(e^*)$. Since $e^*$ is the smallest cut-edge and $e'$ is one of the cut-edges, we have $w(e') \geq w(e^*)$. This gives $w(E') \leq w(E^*)$. As $E^*$ is a MST, we must have that $E'$ is a MST too. As now both $E_1$ and $e^*$ are in $E'$, we conclude that $E_1 \cup \{e^*\}$ is a partial MST (and the MST is $E'$). □

## Correctness of Prim's Algorithm

We now use above cut-property to prove the correctness of Prim's algorithm. Recall that, Prim's algorithm maintains $E_1$ and $S$, where all vertices in $S$ are connected by edges in $E_1$. It iteratively picks $|V| - 1$ edges;

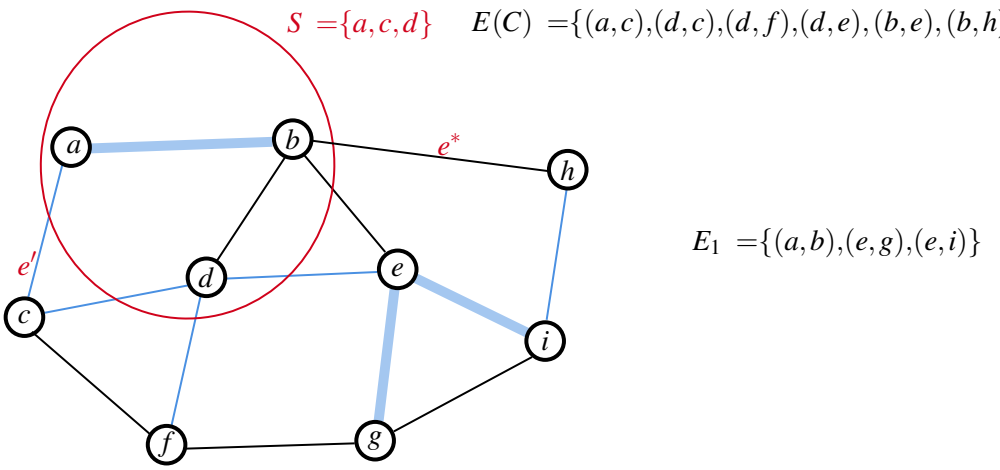



Figure 3: Illustrating the proof of the cut-property. The MST $E^*$ is shown with blue edges (including both thin and thick ones).

in each iteration, it picks the *smallest* edge that connects to a new vertex—the edge picked is *exactly* the smallest cut-edge w.r.t. cut $(S, V \setminus S)$, as cut-edges are those that connect to *new* vertices. A direct application of the cut-property shows that $E_1$ is always a partial MST. Hence, when $|E_1| = |V| - 1$, $E_1$ becomes a MST. This proves the correctness of Prim's algorithm. See Figure 4.

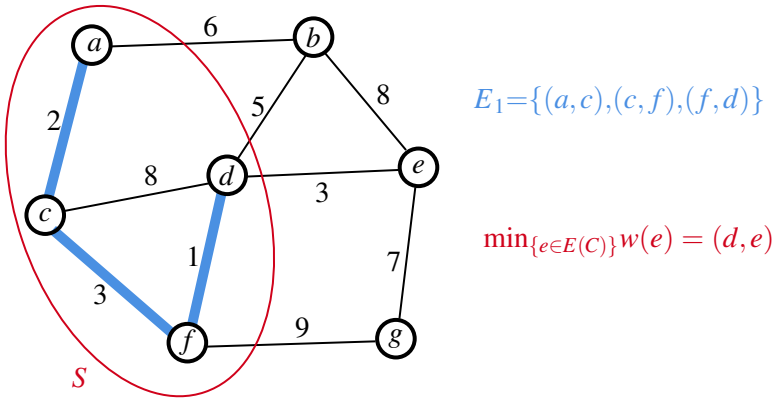



Figure 4: Illustrating the correctness of the Prim's algorithm using cut-property: the next edge picked by the Prim's algorithm is exactly the smallest cut-edge w.r.t. $(S, V \setminus S)$.

## Correctness of Kruskal's Algorithm

Recall that, the Kruskal's algorithm iteratively pick the smallest edge that doesn't produce a cycle, until $E_1$ reaches $|V| - 1$ edges. Again, at any time of the algorithm, $E_1$ partitions all vertices into connected components $\{S_1, S_2, \cdots, S_k\}$; see Figure 5. Note that, adding an edge $e$ that doesn't produce a cycle if and only if edge $e$ spans two components (i.e., not connect two vertices within the same component). So, in each iteration, Kruskal's algorithm simply picks the smallest edge among these that span all pairs of components. Without loss of generality, assume the smallest edge $e^* = (u, v)$ is between $S_i$ and $S_j$, i.e., $u \in S_i$ and $v \in S_j$. We then define cut $C = (S_i, V \setminus S_i)$. Clearly $e^*$ is also the smallest cut-edge w.r.t. $C$, because cut-edges w.r.t. $C$ is a subset of all "spanning edges" while $e^*$ is the smallest spanning edge by the algorithm. Hence, applying

the cut-property here proves that $E_1 \cup \{e^*\}$ is a partial MST too.

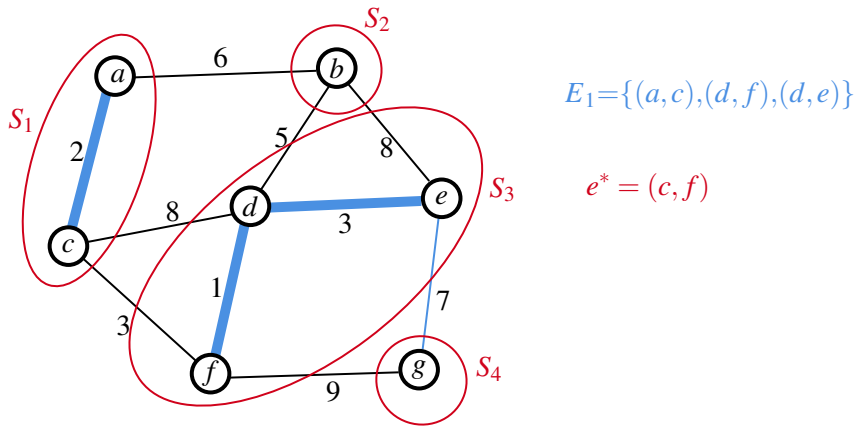



Figure 5: Illustrating the correctness of the Kruskal's algorithm using cut-property: the smallest spanning edge $e^*$ picked by the algorithm is also the smallest cut-edge w.r.t. cut $(S_i, V \setminus S_i)$ where $S_i = S_1$ (or $S_i = S_2$).

# Lecture 28 Disjoint-Set and Its Use in Kruskal's Algorithm

## Disjoint-Set

A *disjoint-set*, also called *union-find*, is an efficient data structure to store *a collection of disjoint sets*. In each set, one of the elements is designated as the *representative* of this set. The representative element serves as the identity of the set. A disjoint-set data structure supports the following operations.

1. *make-set (x)*, creates a new set with $x$ being the only element.
2. *find (x)*, find the set that includes element $x$, by returning the representative of the set.
3. *union (x, y)*, merge the two sets that include $x$ and $y$.

Note that we can simply decide if two elements $x$ and $y$ are in the same set by using above *find* function: two elements are in the same set if and only if they correspond to the same representative, i.e., *find (x)* = *find (y)*.

We use a tree to store a set: elements of a set is stored in the nodes of a tree. The collection of sets is stored with a *forest* (i.e., a collection of trees). The reason we use trees to store sets is that, the *union* operation can be easily implemented by simply merging two trees. We also store the *height* of each subtree. (The height of a subtree rooted at node $v$ is defined as the length of the longest path from $v$ to a leaf node.) The reason we store heights is to maintain a *balanced* tree as much as possible—we don't want the tree to be too tall, as the running time of *find* operation is proportional to the height of the tree (see below). An implementation of the *node* class of a tree will look like this (in C++):

```
class node {
    void* data;  // storing the actual data
    int height;  // storing the height of the subtree rooted at this node
    node* parent;  // pointer to the parent node of this node
};
```

We define the root of each tree always being the representative of the corresponding set. And we always use a pointer that points to itself, i.e., $x.parent = x$, to label that a node is a root. Below we give the pseudo-code for these 3 operations. We assume that elements ($x$, $y$, etc, used below) are with above *node* type.

```
function make-set (x)
    x.height = 1;
    x.parent = x;
end;
```

The *find* operation traverses the path following the parent pointers until reaching the root. Note that the running time of this procedure is propotional to the height of the tree.

```
function find (x)
    while (x.parent ≠ x)
        x = x.parent;
    end;
    return x;
end;
```

The union operation first find the roots of the two trees that contains $x$ and $y$ respectively, and then merge them by hanging one of the tree under the root of the other tree. Notice that we want the tree as shorter as

possible; hence we always hang the shorter tree under the taller tree (break tie arbitrarily when equal). Note too that if the two trees have different heights, after merging all heights remain the same; if the two trees have the same height, then after merging the height of the new root will be increased by 1. See Figure 1.

```
function union (x, y)
    r_x = find(x);
    r_y = find(y);
    if (r_x = r_y): return;  // x and y already in the same tree
    if (r_x.height < r_y.height)
        r_x.parent = r_y
    else if (r_x.height > r_y.height)
        r_y.parent = r_x
    else
        r_x.parent = r_y
        r_y.height = r_y.height + 1
    end
end;
```

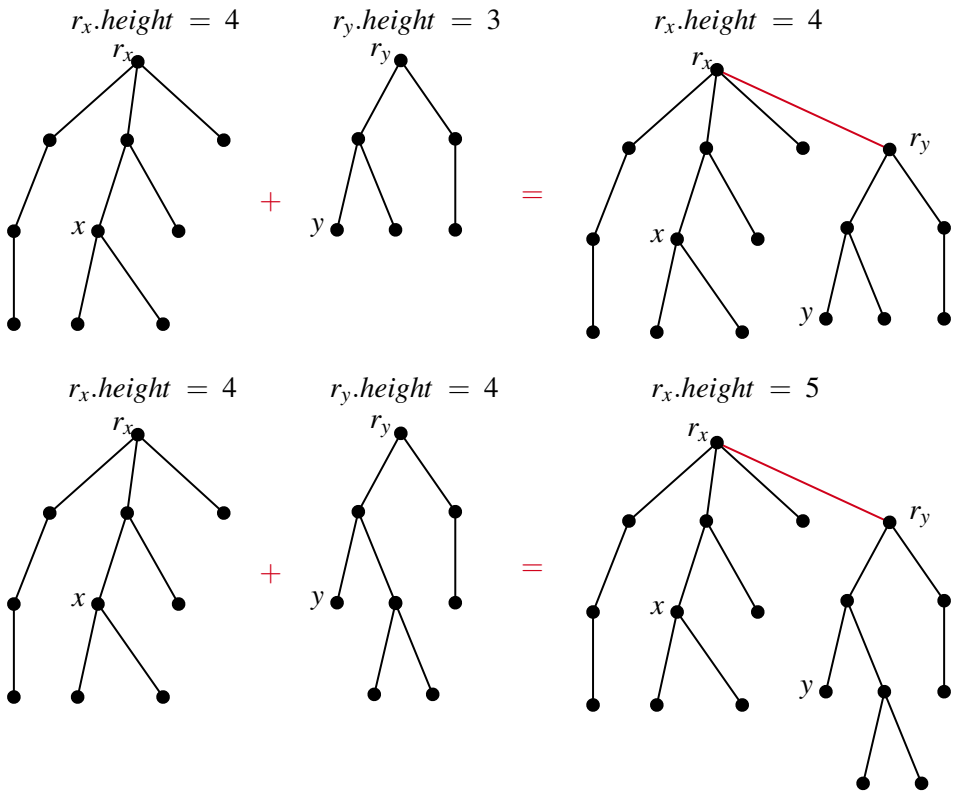



Figure 1: Upper figures: union two trees with different heights: shorter tree will be hung under the root of the taller tree, and all heights keep the same. Lower figures: union two trees with identical heights: the height of the new root will be increased by 1.

## Running Time

We now show that the above balancing procedure guarantees that, the height of any tree is $\Theta(\log n)$.

**Fact 1.** Let $T$ be a tree with $n$ nodes. Then $h(T) \leq 1 + \log_2 n$, where $h(T)$ denotes the height of $T$.

*Proof.* The *make-set* creates a new tree with a single node, and clearly it holds above fact. The *find* operation queries but doesn't change anything. We now show the *union* holds above fact too. We prove it by induction: suppose the before merging $T_1$ and $T_2$ with $n_1$ and $n_2$ nodes respectively, we have $h(T_1) \leq 1 + \log n_1$ and $h(T_2) \leq 1 + \log n_2$. Note that after mering we have $n = n_1 + n_2$. Consider the 3 cases in the union procedure. If $h(T_1) < h(T_2)$, then $h(T) = h(T_2) \leq 1 + \log n_2 \leq 1 + \log n$, as desired. The same for $h(T_1) > h(T_2)$. Now consider $h(T_1) = h(T_2)$. In this case $h(T) = h(T_1) + 1 = h(T_2) + 1$. We can write $2h(T) = 2 + h(T_1) + h(T_2) \leq 4 + \log n_1 + \log n_2 = 4 + \log n_1 n_2 \leq 4 + \log((n_1 + n_2)/2)^2 = 4 + 2\log(n/2) = 2(1 + \log n)$. □

Hence, the *make-set* takes $\Theta(1)$ time; the *find* takes $\Theta(\log n)$ time; the *union* takes $\Theta(\log n)$ time as well (it is dominated by the *find* procedure; the remaining of *union* after calling find takes $\Theta(1)$ time).

## Complete Kruskal's Algorithm

Recall that, in the Kruskal's algorithm, the vertices are partitioned into a collection of connected components by $E_1$, and adding an edge $e = (u, v)$ to $E_1$ doesn't produce a cycle if and only if $u$ and $v$ are in different components. Hence, we can use a disjoint-set data structure to maintain the connected components formed by $E_1$: each set here corresponds to a component. When examining if an edge $e = (u, v)$ leads to a cycle in the algorithm, we simply query if $find(u)$ equals to $find(v)$. If they are not, then $e$ can be safely added, and we can then simply call $union(u, v)$ to update the disjoint-set to reflect that the two components are merged into one by edge $e$.

```
Algorithm Kruskal's (G = (V,E), w(e) ∀e ∈ E)
    sort all edges in E by weights in ascending order;
    E_1 = ∅;
    for each v ∈ V: call make-set (v); // as E_1 = ∅, each vertex forms a set of its own
    for each e = (u,v) ∈ E following above ordering
        r_u = find(u)
        r_v = find(v)
        if (r_u ≠ r_v)
            E_1 = E_1 ∪ {e}
            union(r_u, r_v)
        end;
    end;
end;
```

Sorting takes $\Theta(|E|\log|E|) = \Theta(|E|\log|V|)$ time (note that $\log|E| \leq \log|V|^2 = 2\log|V|$). Examining all edges takes $\Theta(|E|\log|V|)$ too. So the entire algorithm, with this implementation of disjoint-set, takes $\Theta(|E|\log|V|)$ time.

# Lecture 29 Interval Scheduling Problem

Given an array of *intervals* $A[1 \cdots n]$, where $A[i] = (s(i), t(i))$ is an interval represented with *start-time* $s(i)$ and *finish-time* $t(i)$, we seek a *compatible* subset $X \subset A$ such that $|X|$ is maximized. Here, $X$ is *compatible* is defined as that intervals in $X$ are disjoint (i.e., any two intervals in $X$ don't overlap). See Figure 1.

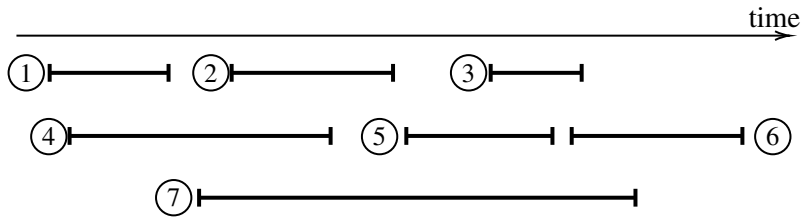



Figure 1: An instance of interval scheduling problem. The optimal solution includes 4 intervals $\{1,2,5,6\}$.

Let's design a greedy algorithm for above interval scheduling problem. Recall that a greedy algorithm iteratively makes *local decisions*, until reaches a complete solution. In particular, for this problem a greedy algorithm follows the following framework.

```
Algorithm greedy-framework (intervals A[1 ··· n])
    init X = ∅;
    while (true)
        greedily pick a (new) interval x ∈ A that doesn't conflict with intervals in X and add it to X;
        the algorithm terminates and returns X if such x cannot be found;
    end;
end;
```

Above procedure is a framework, as how to *greedily pick a new interval that is compatible with X* is not specified. A common practice in order to design a correct (i.e., optimal) greedy algorithm is exploring different greedy strategies. For each strategy, we first try to design a counter-example to show it doesn't work; if such an instance is hard to construct, then it's a promising direction and we then try to prove that it is optimal. For this interval scheduling problem, we can explore the following strategies:

1. In each iteration, pick the interval with smallest length that is compatible with $X$. In fact, Figure 1 is an counter-example, where this greedy strategy finds intervals $\{3,1,2\}$.

2. In each iteration, pick the interval with fewest conflicts that is compatible with $X$. You may notice that this strategy actually finds the optimal solution on Figure 1. But it cannot guarantee optimality on all instances. Can you design an counter-example? Figure 4.1 (c) of the textbook [KT], page 117, gives such a counter-example.

3. In each iteration, pick the interval with earliest start-time that is compatible with $X$. In fact, Figure 1 again is an counter-example, where this greedy strategy finds intervals $\{1,7\}$.

4. In each iteration, pick the interval with earliest finish-time that is compatible with $X$. You may notice that this strategy also finds the optimal solution on Figure 1. In fact, this strategy is indeed optimal (see the formal proof below).

## Correctness

We now show that, the greedy strategy of picking the (compatible) interval with earliest finish-time leads to an optimal greedy algorithm. First, note that this greedy framework guarantees that $X$ is compatible (even if using the first 3 strategies), as any newly added interval doesn't overlap with existings ones in $X$.

Now we show that the algorithm is optimal, i.e., $|X|$ is maximized. Assume that $X = \{A[x_1], A[x_2], \cdots, A[x_k]\}$. We also assume that these $k$ intervals are sorted by their finish-time, i.e., $t(x_1) \leq t(x_2) \leq \cdots \leq t(x_k)$. Assume that $X^* = \{A[x_1^*], A[x_2^*], \cdots, A[x_m^*]\}$ be an optimal solution (with $m$ intervals). We also assume that these $m$ intervals are sorted by their finish-time, i.e., $t(x_1^*) \leq t(x_2^*) \leq \cdots \leq t(x_m^*)$.

Clearly $k \leq m$, as we assume $X^*$ is the optimal solution. The ultimate goal is to prove that $k = m$, i.e., the greedy algorithm always finds the same number of intervals with the optimal solution. In order to achieve it, we first prove a more informative fact, which shows that the greedy solution always *stays ahead* of the optimal solution. Specifically, the finish-time of the $i$-th interval in the greedy solution is always no latter than the finish-time of the $i$-th interval of the optimal solution, formally stated below.

**Fact 1.** For any $1 \leq i \leq k$, we have $t(x_i) \leq t(x_i^*)$.

*Proof.* We prove it by induction. The base case is to show $t(x_1) \leq t(x_1^*)$. This is obvious, as the first interval picked by the greedy algorithm is always the one with smallest finish-time.

We now show the inductive step. Suppose $t(x_j) \leq t(x_j^*)$ for all $1 \leq j \leq i$. We aim to prove $t(x_{i+1}) \leq t(x_{i+1}^*)$. See Figure 2. We can write $t(x_i) \leq t(x_i^*) \leq s(x_{i+1}^*)$. This shows that, $A[x_{i+1}^*]$ does not conflict with any of these $\{A[x_1], A[x_2], \cdots, A[x_i]\}$. Hence, when greedy algorithm picks the $(i+1)$-th interval, $A[x_{i+1}^*]$ is one of the legitimate candidates. As the greedy algorithm always pick the one with earliest finish-time (among all legitimate ones), either $A[x_{i+1}^*]$ is picked, or another one with even smaller finish-time (than $A[x_{i+1}^*]$) is picked; in either case, we have $t(x_{i+1}) \leq t(x_{i+1}^*)$, as desired. □

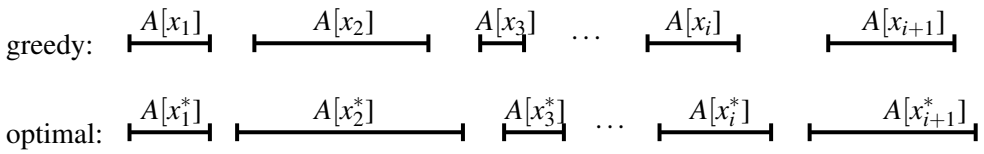



Figure 2: Illustrating of proving Fact 1.

**Fact 2.** We have $k = m$.

*Proof.* We prove it by contradiction. See Figure 3. Suppose conversely that $m \geq k+1$. Using the same reasong in proving Fact 1, we know that after picking $\{A[x_1], A[x_2], \cdots, A[x_k]\}$ in the greedy algorithm, $A[x_{i+1}^*]$ remains legitimate, as it does not overlap with any of these $k$ intervals that are already picked. Hence, the greedy algorithm won't terminate and will continue to pick at least one more interval. This contradicts to the assumption that the greedy solution includes $k$ intervals. □

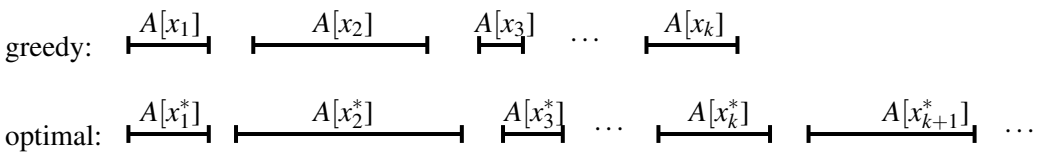



Figure 3: Illustrating of proving Fact 2.

## The Complete Algorithm

How to efficiently pick the interval with earliest finish-time that is compatible with existing ones (i.e., those in $X$)? We can sort all intervals by their finish-time, examine them one-by-one following this order, and include it when it does not conflict with those in $X$—a similar approach is used in the Kruskal's algorithm. Then in each itertion, how to decide if the current interval is compatible with $X$? A naive solution is to compare it with all intervals in $X$, but this will take linear time and the resulting algorithm will take $O(n^2)$ time. In fact, we can do it faster. As all intervals are now sorted by their finish-time, we have that the current interval can be safely added if and only if its start-time is larger or equal to the finish-time of the previously added interval. Hence, we simply maintain the last interval that is added to $X$, and we can use a single comparison, which takes constant time, to decide if the current one is compatible with $X$ or not.

The pseudo-code is given below. The for-loop takes linear time. The entire algorithm takes $O(n \log n)$ time, as it is dominated by sorting. An example of running this algorithm follows.

```
Algorithm greedy (intervals A[1 ··· n])
    sort A by their finish-time;
    init X = {A[1]};
    let j = 1; // maintain the previously added interval
    for i = 2 → n
        if (s(i) ≥ t(j))
            pick A[i] and add it to X;
            j = i;
        end;
    end;
    return X;
end;
```

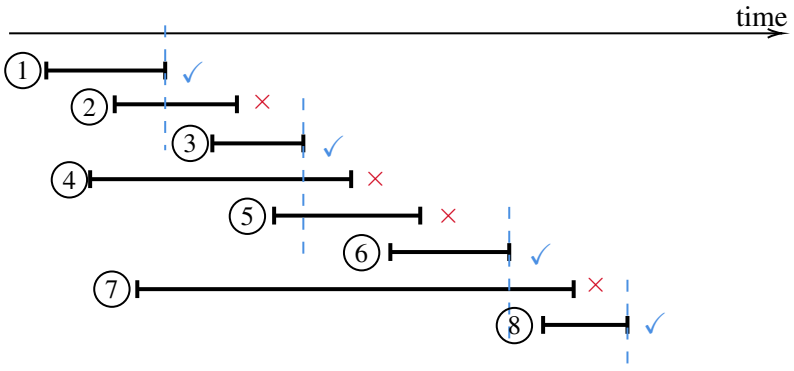



Figure 4: An example of running above algorithm.

# Lecture 30 Weighted Interval Scheduling Problem

## The Problem

Given an array of *intervals* $A[1 \cdots n]$, where $A[i] = (s(i), t(i))$ is an interval represented with *start-time* $s(i)$ and *finish-time* $t(i)$, and value $v(i)$ associated with $A[i]$, we seek a *compatible* subset $X \subset A$ such that the total value of the intervals in $X$, i.e., $\sum_{A[i] \in X} v(i)$, is maximized. Again, $X$ is *compatible* is defined as that intervals in $X$ are disjoint (i.e., any two intervals in $X$ don't overlap). See an instance in Figure 1. Clearly, this weighted interval scheduling problem is a generalization of the interval scheduling problem introduced in last lecture, for which each interval is associated with unit value, i.e., $v(i) = 1, 1 \leq i \leq n$.

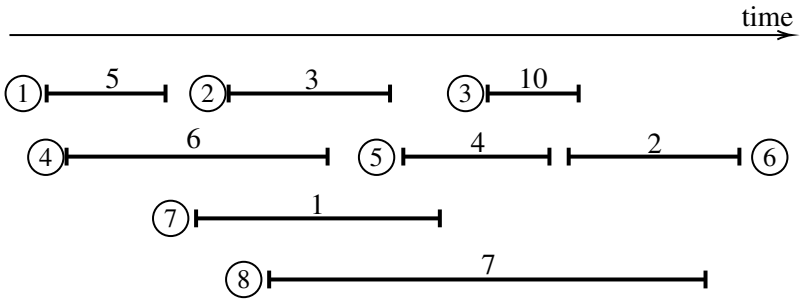



Figure 1: An instance of the weighted interval scheduling problem. The optimal solution includes 3 intervals $\{1,2,3\}$, and the (optimal) total value is 18.

Can we also design a greedy algorithm like we did for the interval scheduling problem? Let's try. Again the framework is to maintain a compatible subset $X$, and in each iteration we add one more interval that is compatible with $X$. We now explore different greedy strategies.

1. In each iteration, pick the interval with the earliest finish-time that is compatible with $X$—which has been proved optimal for the unit-value version. However, this strategy isn't optimal here—Figure 1 is an counter-example, where this greedy strategy finds intervals $\{1,2,5,6\}$ with a total value of 14.

2. In each iteration, pick the interval with largest value that is compatible with $X$. This strategy isn't optimal either. Figure 1 is again a counter-example, where this greedy strategy finds intervals $\{3,4\}$ with a total value of 16.

As you can see, all these greedy strategies fail. In fact, it's unlikely to design a greedy algorithm for this problem. We therefore seek other techniques. Below we will design a dynamic programming algorithm.

## A Dynamic Programming Algorithm

Recall that in order to design a DP algorithm, we need to figure out its *optimal substructure*. Does this problem satisfy the optimal substructure property? Let's check Figure 1 to try to get some clue. In that example, the *optimal solution* is a set of intervals, namely $X = \{A[1], A[2], A[3]\}$. Naturally, the *substructure* of this optimal solustion is one of its subset, say, $X' = \{A[1], A[2]\}$, by removing one element $A[3]$. Then we ask: is $X'$ the *optimal solution* of certain subproblem? In fact, yes! If we exclude $A[3]$ and all intervals that conflict with $A[3]$ from $A$, the remaining will be $A' = \{A[1], A[2], A[4], A[7]\}$, and $X'$ is the *optimal solution* of $A'$. (The proof is again a replacing argument: if there exists a better solution (than $X'$) of $A'$, then we can combine it with $A[3]$ to get a better solution of $A$.) So, this problem indeed satisfies an optimal substructure property, and therefore a DP algorithm can be designed.

In fact, a direct use of above optimal substructure leads to exponential number of subproblems (as we need to define a subproblem for *every* possible subset of $A$). We need an ordering to reduce the number of subproblems. Again, we can sort all intervals by their finish-time (and rename them; now we assume that all intervals in $A$ are sorted; see Figure 2). Now let's check the optimal substructure again with sorted intervals. The optimal solution is $X = \{A[1], A[3], A[6]\}$. Consider its substructure $X' = \{A[1], A[3]\}$ by removing the last one in the order, i.e., $A[6]$. We have that $X'$ is the optimal solution of $A[1 \cdots 4]$. (Again, try the replacement-argument to prove it). And why $A[1 \cdots 4]$? This is because these four intervals do not overlap with $A[6]$ but $A[5]$ does. In other words, $A[4]$ is the *rightmost interval before* $A[6]$ *that doesn't conflict with* $A[6]$. In fact, if $A[4]$ don't overlap with $A[6]$, i.e, $t(4) \leq s(6)$, then all intervals to the left of $A[4]$ will not overlap with $A[6]$ as $t(1) \leq t(2) \leq t(3) \leq t(4)$.

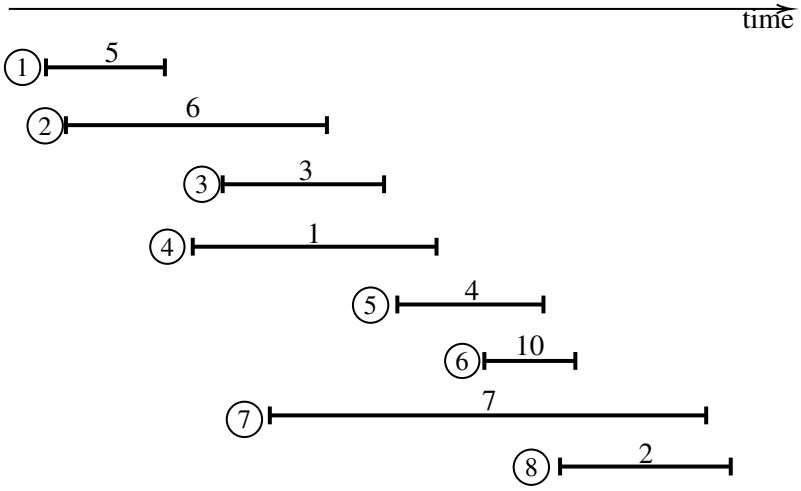



Figure 2: An instance of the weighted interval scheduling problem. The optimal solution includes 3 intervals $\{1,2,3\}$, and the (optimal) total value is 18.

Above analysis suggests we define subproblem over $A[1 \cdots k]$ for every $1 \leq k \leq n$. Formally, we define $F(k)$ be the achievable total value over the first $k$ intervals, i.e., $A[1 \cdots k]$. We now develop a recursion. Again, we try to enumerate all possiblities of the last step. Here, that's whether $A[k]$ is picked or not. Suppose that, the optimal solution (of $A[1 \cdots k]$) doesn't include $A[k]$; in this case we have $F(k) = F(k-1)$. Suppose that, the optimal solution includes $A[k]$; in this case we have $F(k) = v(k) + F(pre[k])$. Here, we use term $pre[k]$ to represent the *rightmost interval before* $A[k]$ *that doesn't conflict with* $A[k]$. Formally, $pre[k] = \max\{j < k \mid t(j) \leq s(k)\}$. Combined, we have the following recursion.

$$F(k) = \max \begin{cases} F(k-1) \\ F(pre[k]) + v(k) \end{cases}$$

One question remain: how to calculate $pre[k]$? One straitforward approach is to compare $A[k]$ with all intervals before $A[k]$ in reverse order, i.e., $A[k-1], A[k-2], \cdots, A[1]$; the first $A[j]$ satisfies that $t(j) \leq s(k)$ will be $pre[k]$. This clearly takes linear time. In fact, we can use *binary search* to get a $O(\log n)$-time algorithm. This works because, all intervals in $A$ are already sorted by their finish-time.

Specifically, we define a recursive function *binary-search (*$A[1 \cdots n]$*, k, a, b)* with 4 parameters: $A[1 \cdots n]$ is the sorted $n$ intervals, $k$ specifies that we aim to calculate $pre[k]$, $a$ and $b$ specifies the range of $A$ where we will search for. Its pseudo-code is given below. Clearly, this binary-search procedure runs in $O(\log n)$ time. To find $pre[k]$, we actually call: *binary-search (A, k,* 1*,* $k-1$*)*.

```
function binary-search (A[1···n], k, a, b)
    m = (a+b)/2;
    if (t(m) ≤ s(k) ≤ t(m+1))  // pre[k] = m
        return m;
    else if (t(m) > s(k))  // pre[k] must be in [a···m]
        return binary-search (A, k, a, m);
    else if (s(k) > t(m+1))   // pre[k] must be in [m+1···b]
        return binary-search (A, k, m+1, b);
    else
        return 0;
    end;
end;
```

The complete DP algorithm for solving the weighted interval scheduling problem is given below. Its running time is $O(n\log n)$: the sorting step takes $\Theta(n\log n)$, and for each $k$, the binary-search takes $O(\log n)$, and hence the entire for-loop takes $O(n\log n)$ as well.

```
Algorithm DP-weighted (intervals A[1···n], v(i), 1 ≤ i ≤ n)
    sort A by their finish-time;
    init array F of size n+1 and set F[0] = 0;
    for k = 1 → n
        pre[k] = binary-search (A, k, 1, k−1);
        F[k] = max{F[k−1], F[pre[k]] + v(k)};
    end;
    return F[n];
end;
```

# Lecture 31 Maximum Flow and Minimum Cut

A *network* (also called *flow network*) models an abstracted flow (traffic, data, etc) transmits from an origin to a destination via a graphical structure. Formally, a (flow) network is a tuple $(G = (V,E), s, t, c(\cdot))$, where

1. $G = (V,E)$ is a directed graph;
2. $s \in V$ is a source vertex of $G$ (i.e., the in-degree of $s$ is 0), from which the flow is generated;
3. $t \in V$ is a sink vertex of $G$ (i.e., the out-degree of $t$ is 0), at which the flow is absorbed;
4. $c : E \to \mathbb{R}^+$, in which $c(e)$ represents the *capacity* of edge $e \in E$, which models the limit of the flow that edge $e$ can carry.

An *s-t flow* of network $(G = (V,E), s, t, c(\cdot))$ describes how the flow is transmitted from $s$ to $t$ via the graph $G$ under the capacity constraints. Formally, an $s$-$t$ flow $f$ is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the following two conditions:

1. For every $e \in E$, $0 \leq f(e) \leq c(e)$; this is called the *capacity condition*;
2. For every vertex $v \in V \setminus \{s,t\}$, $\sum_{e \in I(v)} f(e) = \sum_{e \in O(v)} f(e)$, where $I(v) := \{(u,v) \in E \mid u \in V\}$ represents the set of in-edges of $v$, and $O(v) := \{(v,w) \in E \mid w \in V\}$ represents the set of out-edges of $v$. This is called the *conservation condition*.

Intuitively, an $s$-$t$ flow assigns an non-negative value $f(e)$ to edge $e$, representing the amount of flow that is carried by edge $e$; such amount must not exceed the capacity of $e$, i.e., $f(e) \leq c(e)$, for every $e \in E$. All vertices, except the source $s$ that generates the flow and the sink that absorbs the flow, redistribute the flow (instead of repositing any flow); therefore, for each vertex $v \in V \setminus \{s,t\}$, the total amount of flow that enters $v$ equals to the total amount of flow that leaves $v$. See Figure 1.

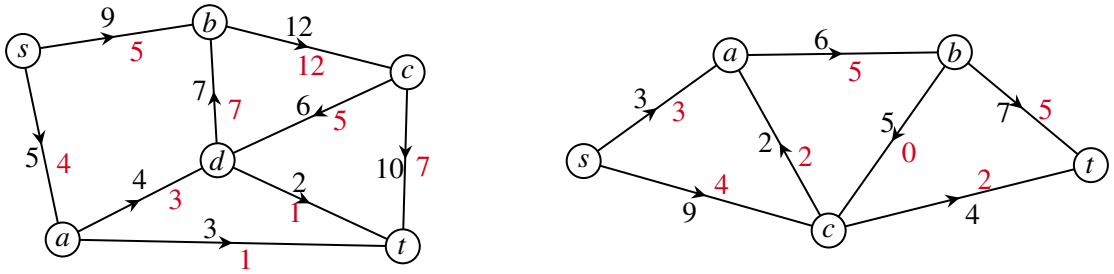



Figure 1: Two examples of networks and $s$-$t$ flows. The capacities are marked as black numbers next to edges; the flow values are marked as red numbers next to edges. The value of the flow in the left exampe is 9; the value of the flow in the right exampe is 7;

The *value* of an $s$-$t$ flow $f$, denoted as $|f|$, is defined as the total amount of flow that is generated from source vertex $s$: $|f| := \sum_{e \in O(s)} f(e)$. Intuitively, the total amount of flow that is generated from $s$ eventually must be fully absorbed by sink $t$, as all internal vertex (i.e., $V \setminus \{s,t\}$) does not keep any flow. Therefore, we must have that $|f| = \sum_{e \in I(t)} f(e)$. This property is also a direct consequence of the conservation property. Below we formally state and prove it.

**Fact 1.** We have $\sum_{e \in O(s)} f(e) = \sum_{e \in I(t)} f(e)$.

*Proof.* According to the conservation condition: $\sum_{e\in I(v)} f(e) = \sum_{e\in O(v)} f(e)$, for every vertex $v \in V \setminus \{s,t\}$. We sum up both sides over all $v \in V \setminus \{s,t\}$: we have $\sum_{v\in V\setminus\{s,t\}} \sum_{e\in I(v)} f(e) = \sum_{v\in V\setminus\{s,t\}} \sum_{e\in O(v)} f(e)$.

We now break down the left side $L := \sum_{v\in V\setminus\{s,t\}} \sum_{e\in I(v)} f(e) = \sum_{v\in V} \sum_{e\in I(v)} f(e) - \sum_{e\in I(s)} f(e) - \sum_{e\in I(t)} f(e)$. Notice also that $\sum_{v\in V} \sum_{e\in I(v)} f(e) = \sum_{e\in E} f(e)$. Since $s$ is a source vertex, we have $I(s) = \emptyset$, and therefore $\sum_{e\in I(s)} f(e) = 0$. Hence, $L = \sum_{e\in E} f(e) - \sum_{e\in I(t)} f(e)$.

Similarly, the right side $R := \sum_{v\in V\setminus\{s,t\}} \sum_{e\in O(v)} f(e) = \sum_{v\in V} \sum_{e\in O(v)} f(e) - \sum_{e\in O(s)} f(e) - \sum_{e\in O(t)} f(e)$. Again we have $\sum_{v\in V} \sum_{e\in O(v)} f(e) = \sum_{e\in E} f(e)$. Since $t$ is a sink, we have $O(t) = \emptyset$, and therefore $\sum_{e\in O(t)} f(e) = 0$. Hence, $R = \sum_{e\in E} f(e) - \sum_{e\in O(s)} f(e)$.

Combining $L = R$ and noticing that $\sum_{e\in E} f(e)$ is shared, we have $\sum_{e\in O(s)} f(e) = \sum_{e\in I(t)} f(e)$, as desired. □

We now define the *maximum-flow problem*: given network $(G = (V,E), s, t, c(\cdot))$, to find an $s$-$t$ flow $f$ such that $|f|$ is maximized.

See Figure 2 for an example, which shows a flow with value of 9. This flow has a larger value than the flow in Figure 1 (right panel, same network, but with a flow of value 7). Can you play with this example to try to obtain a flow with even larger value? Is this flow (of value 9) the maximum flow of the network? If it is, how can we verify it is one maximum flow? Let's answer these questions using $s$-$t$ cut.

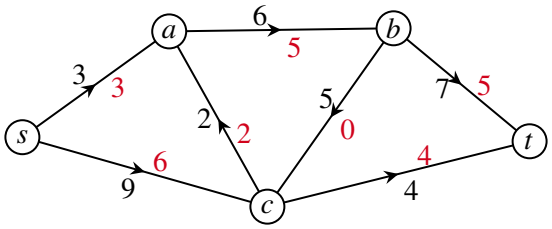



Figure 2: A flow with value 9 (same network with the right panel of Figure 1.

Given a network $G = (V,E)$ with source $s \in V$, sink $t \in V$, and capacity $c(\cdot)$, an *$s$-$t$ cut* of the network is a pair $(S,T)$, where $S \subset V$, $T = V \setminus S$, and that $s \in S$ and $t \in T$. In short, an $s$-$t$ cut of a network is a partition of the vertices that separates the source and the sink.

An $s$-$t$ cut $(S,T)$ of a network also partitions all edges in the graph into four disjoint subsets, described below. We define the first category, i.e., $E(S,T) := \{(u,v) \in E \mid u \in S, v \in T\}$, as the *cut-edges* w.r.t. the $s$-$t$ cut $(S,T)$.

1. Edges that span the cut and point from $s$-side to $t$-side, formally $E(S,T) := \{(u,v) \in E \mid u \in S, v \in T\}$;
2. Edges that span the cut and point from $t$-side to $s$-side, formally $E(T,S) := \{(u,v) \in E \mid u \in T, v \in S\}$;
3. Edges with both end-vertices in $s$-side, formally $E(S,S) := \{(u,v) \in E \mid u \in S, v \in S\}$;
4. Edges with both end-vertices in $t$-side, formally $E(T,T) := \{(u,v) \in E \mid u \in T, v \in T\}$.

Let $(S,T)$ be an $s$-$t$ cut of a network $G = (V,E)$ with source $s$, sink $t$, and capacity $c(e)$ for any $e \in E$. We define the *capacity* of $(S,T)$, denoted as $c(S,T)$, as the sum of the capacities of its cut-edges. Formally as $c(S,T) := \sum_{e\in E(S,T)} c(e)$. See Figure 3.

We now introduce the so-called *minimum-cut problem*: given network $G = (V,E)$ with source $s$, sink $t$ and capacity $c(e)$ for any $e \in E$, we seek an $s$-$t$ cut $(S,T)$ such that its capacity $c(S,T)$ is minimized.

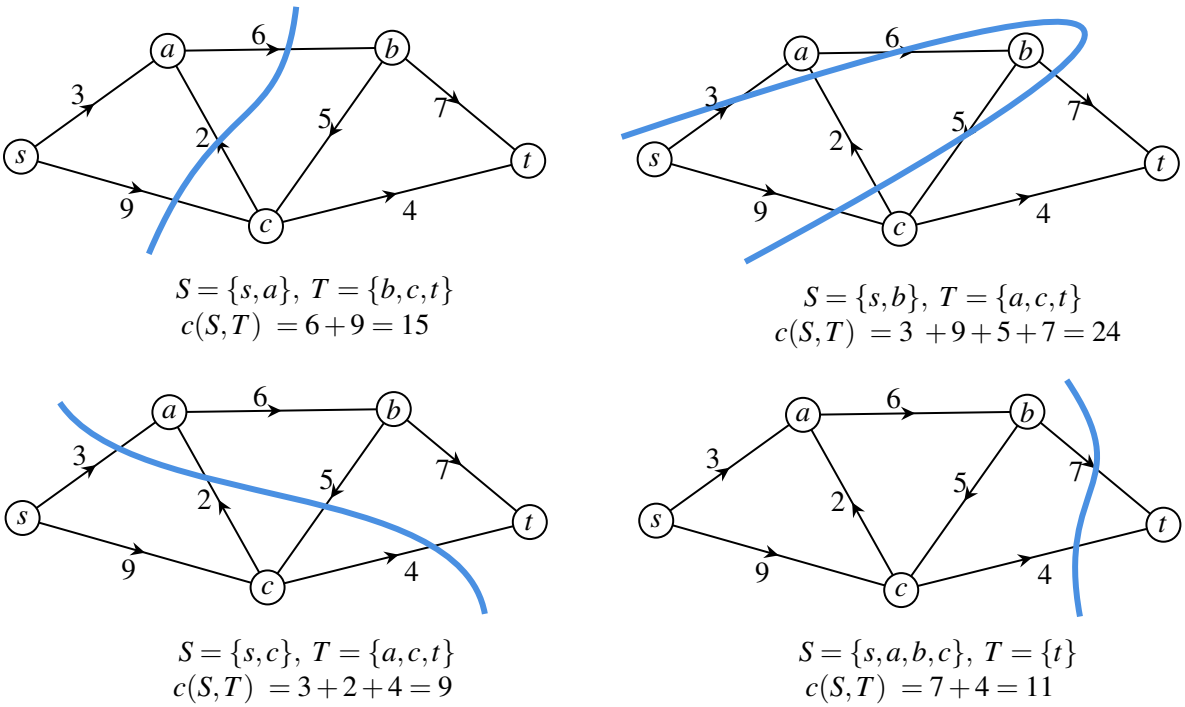



Figure 3: Four different *s-t* cuts and their capacities of the same network.

Note that, the definition of *s-t* cut, capacity of an *s-t* cut, and above minimum-cut problem, are completely independent of flow. Note too that, the input of the maximum-flow problem and the minimum-cut problem is the same: a network (consisting of a directed graph, source and sink, and edge capacities).

Although the maximum-flow problem and the minimum-cut problem are defined independently, they can be solved by the same algorithm (for example, the Ford-Fulkson algorithm) and are connected by an elegant theorem (i.e., the max-flow min-cut theorem). Below, we first show half of this theorem. Later on we will introduce the Ford-Fulkson algorithm and use it to prove the other other half of the theorem.

We now show that, the capacity of *any s-t* cut of a network gives an *upper bound* of the value of *any s-t* flow of the same network, formally stated below.

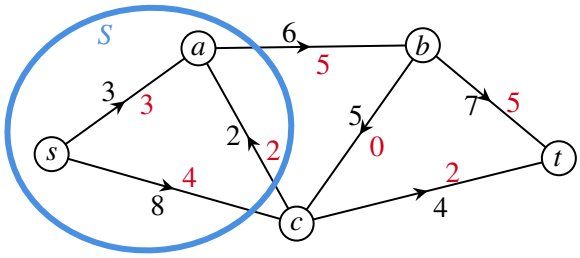
**Claim 1.** Let $(G = (V,E), s, t, c(\cdot))$ be a network. Let $f$ be an arbitrary *s-t* flow and let $(S,T)$ be an arbitrary *s-t* cut, of this network. We have $|f| \leq c(S,T)$.

Above claim is intuitive to understand: an *s-t* flow $f$ transfer $|f|$ units of flow from source $s$ to sink $t$; since an *s-t* cut $(S,T)$ separates $s$ and $t$, so the flow, all generated from $s$, must cross the cut in order to reach $t$, and such "crossing" must use the cut-edges. Hence, the total amount of the flow that can be transferred, i.e., the value of $f$, is limitted by the total capacities of the cut-edges, i.e., the capacity of the *s-t* cut. See Figure 4.

In order to formally prove above claim, we first show that, the value of an *s-t* flow $f$ can also be calculated by examining the flow carried by the "spanning edges" of any *s-t* cut. Formally,

**Fact 2.** Let $f$ be an arbitrary *s-t* flow and let $(S,T)$ be an arbitrary *s-t* cut, of network $(G = (V,E), s, t, c(\cdot))$. We have $|f| = \sum_{e \in E(S,T)} f(e) - \sum_{e \in E(T,S)} f(e)$.

Above fact is also intuitive to understand: $S$ is a subset that includes $s$ but excludes $t$. See Figure 4. When




$S = \{s,a\},\ T = \{b,c,t\}$
$c(S,T) = 6+8 = 14$
$|f| = 7$

Figure 4: An example showing Claim 1 and Fact 1.

thinking $S$ as a whole, the total amount of flow generated by $S$ is equal to $|f|$, as only $s$ generates flow. On the other hand, this amount of flow emitted from $S$ can also be calcuated by summing up the flow leaves $S$, i.e., $\sum_{e\in E(S,T)} f(e)$, and subtracting the amount of the flow that enters back to $S$, i.e., $\sum_{e\in E(T,S)} f(e)$.

We now formally prove Fact 1. Similar to the proof of Fact 1 of Lecture 36, this proof also simply applies the conservation property.

*Proof of Fact 1.* According to conservation property, for any $v \in S\setminus\{s\}$, we have $\sum_{e\in I(v)} f(e) = \sum_{e\in O(v)} f(e)$. We sum up both sides over all $v \in S\setminus\{s\}$: we have $\sum_{v\in S\setminus\{s\}} \sum_{e\in I(v)} f(e) = \sum_{v\in S\setminus\{s\}} \sum_{e\in O(v)} f(e)$. Note that $I(s) = \emptyset$ as $s$ is a source, so the left side of above equation can be rewritten as $\sum_{v\in S} \sum_{e\in I(v)} f(e)$. Note that $|f| = \sum_{e\in O(s)} f(e)$ by definition, so the right side can be rewritten as $\sum_{v\in S} \sum_{e\in O(v)} f(e) - |f|$. Combined, we have $|f| = \sum_{v\in S} \sum_{e\in O(v)} f(e) - \sum_{v\in S} \sum_{e\in I(v)} f(e)$. Now consider the 4 types of edges partitioned by $(S,T)$ and consider if they are taken into account by the two items i.e., $\sum_{v\in S} \sum_{e\in O(v)} f(e)$ and $\sum_{v\in S} \sum_{e\in I(v)} f(e)$.

1. All edges in $E(S,T)$ appear in the first item, but not the second one.
2. All edges in $E(T,S)$ appear in the second item, but not the first one.
3. All edges in $E(S,S)$ appear in both items (so they cancel out).
4. None of the edges in $E(T,T)$ appear in either item.

Therefore, we have $|f| = \sum_{e\in E(S,T)} f(e) - \sum_{e\in E(T,S)} f(e)$. □

We now formally prove Claim 1. The proof simply combines the capacity condition (and the non-negative property) of an $s$-$t$ flow, Fact 1, and the definition of the capacity of an $s$-$t$ cut.

*Proof of Claim 1.* Since $0 \le f(e) \le c(e)$ for any $e \in E$, based on Fact 1, we can write $|f| = \sum_{e\in E(S,T)} f(e) - \sum_{e\in E(T,S)} f(e) \le \sum_{e\in E(S,T)} c(e) - 0 = c(S,T)$. □

Claim 1 states that capacity of *any* $s$-$t$ cut is larger or equal to the value of *any* $s$-$t$ flow. Consequently, if there exists an $s$-$t$ flow $f$ and an $s$-$t$ cut $(S,T)$ satisfies that $|f| = c(S,T)$, then this $f$ must be a maximum-flow, and this $(S,T)$ must be a minimum-cut. An illustration of this property is given in Figure 5, where each red point represents the value of an $s$-$t$ flow while each blue point represents the capacity of an $s$-$t$ cut. So all red

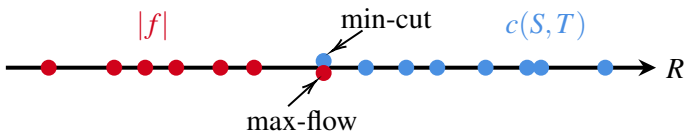
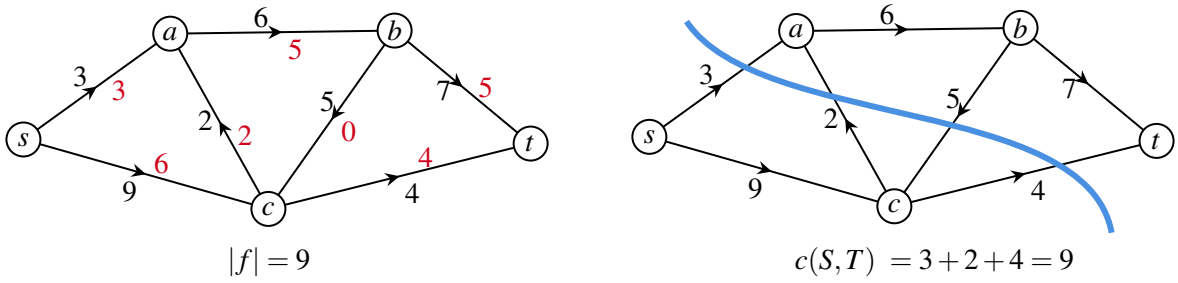



Figure 5: Illustration of the relationship between capacity of any $s$-$t$ cut and value of *any* $s$-$t$ flow.

points lie to the left side of all blue points, and if a red point and a blue point meet, they are maximum-flow and minimum-cut.




$|f| = 9$ $\quad c(S,T) = 3+2+4 = 9$

Figure 6: A maximum-flow $f$ and a minimum cut $(S,T)$, verified by $|f| = c(S,T)$.

In above example, we see a flow $f$ with $|f| = 9$ and an $s$-$t$ cut $(S,T)$ with $c(S,T) = 9$. So both are optimal. See Figure 6.

# Lecture 32 Ford-Fulkson Algorithm and The Max-Flow Min-Cut Theorem

At the end of previous Lecture we gave an example (network) for which there exists a flow $f$ and an $s$-$t$ cut such that $|f| = c(S,T)$. Therefore, both $f$ and $(S,T)$ are optimal. Is this the case for *any* network? That is, for an arbitrary network, does there *always* exist $f$ and $(S,T)$ such that $|f| = c(S,T)$? The answer is yes! (This elegant, surprising result is referred to the max-flow min-cut theorem.) The proof is constructive: we will design an algorithm (i.e., the Ford-Fulkson algorithm) that actually always finds an $s$-$t$ flow $f$ and an $s$-$t$ cut $(S,T)$ that satisfies $|f| = c(S,T)$.

## The Ford-Fulkson Algorithm

The Ford-Fulkson algorithm aims to find a maximum-flow of the given network. It employs the idead of *iterative improving*. The algorithm starts from a trivial flow $f$ with $f(e) = 0$ for every $e \in E$. It then iteratively improve $f$. In each iteration, it finds a path $p$ from $s$ to $t$ in the so-called *residual graph w.r.t. the current flow* $f$, and then improve $f$ by *augmenting* path $p$. The value of $f$ will be increased after each iteration, and the algorithm will terminate when such path cannot be found. Below see the pseudo-code of the Ford-Fulkson's algorithm.

```
Algorithm Ford-Fulkson (G = (V,E), s, t, c)
    init an s-t flow f with f(e) = 0 for any e ∈ E;
    while (true)
        build the residual graph G_f w.r.t. the current flow f;
        find an s-t path p in G_f;
        if such path cannot be found: return f;
        f ← augment(f, p);
    end;
end;
```

**Residual Graph.** The residual graph plays a central role in the Ford-Fulkson algorithm and in proving the max-flow min-cut theorem. Let $f$ be an $s$-$t$ flow of a network $(G = (V,E), s, t, c)$. We denote by $G_f = (V, E_f)$ the residual graph w.r.t. $f$. We emphasize that a residual graph is always associated with (i.e., w.r.t.) a flow of a network. Each edge $e \in E_f$ in the residual graph is also associated with a capacity, denoted as $c_f(e)$. The construction of the residual graph is given below. See Figure 1.

1. The residual graph has the same set of vertices with the network.
2. For each edge $(u,v) \in E$, there are two corresponding edges in $E_f$: the *forward-edge* $(u,v)$ with capacity $c_f(u,v) = c(u,v) - f(u,v)$, and the *backward-edge* $(v,u)$ with capacity $c_f(v,u) = f(u,v)$.

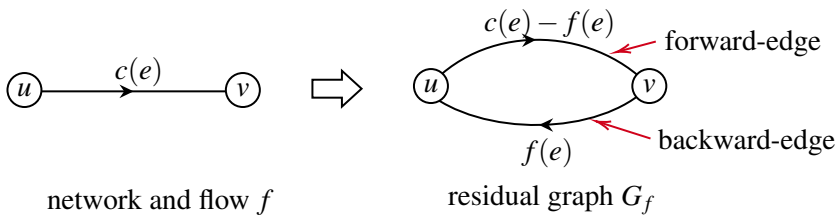
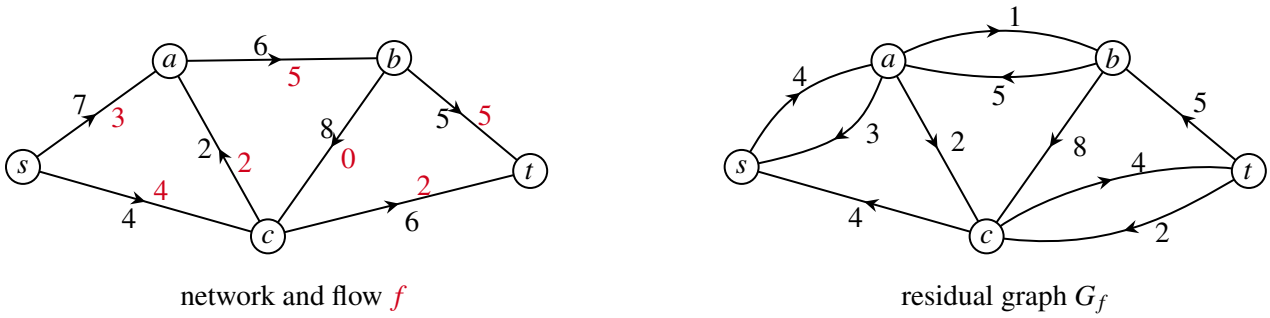



Figure 1: Definition of edges and capacities of the residual graph.

Edges in the residual graph with capacity of 0 will be removed. In other words, we always assume that edges

in the residual graph have positive capacities. Therefore, if an edge $e = (u,v) \in E$ in the network carries a flow of 0, i.e., $f(e) = 0$, then the residual graph only includes the corresponding forward-edge $(u,v)$ with capacity $c_f(u,v) = c(e)$; if an edge $e = (u,v) \in E$ is *saturated*, i.e., $f(e) = c(e)$, then the residual graph only includes the corresponding backward-edge $(v,u)$ with capacity $c_f(v,u) = f(e) = c(e)$. See Figure 2.




network and flow $f$ residual graph $G_f$

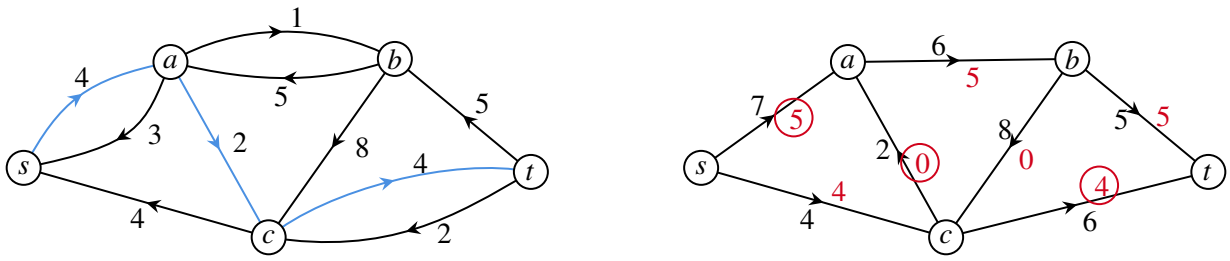
Figure 2: An example of residual graph.

**Finding an $s$-$t$ path in $G_f$.** In each iteration of the Ford-Fulkson algorithm, after building $G_f$ wrt the current flow $f$, it seeks a path $p$ from $s$ to $t$, called an $s$-$t$ path, in residual graph $G_f$. The searching of such $s$-$t$ path can be done by either BFS or DFS, starting from $s$. If such path cannot be found, the algorithm terminates and returns the current flow $f$. Otherwise, it *augments* $p$ to obtain a flow with increased value.

**Augmenting an $s$-$t$ path $p$ in $G_f$.** To augment an $s$-$t$ path $p$ in $G_f$, we first calcualte the bottleneck capacity of $p$, which is defined as the smallest capacity (in the residual graph $G_f$) of all edges in $p$, formally written as $x(p) := \min_{e \in p} c_f(e)$. We then examine each edge $e \in p$, and update the flow of the corresponding edge in the network with the following rule.

1. If $e = (u,v) \in p$ is a forward edge, i.e., $(u,v)$ is in the network, we update $f(u,v) \leftarrow f(u,v) + x(p)$;
2. If $e = (u,v) \in p$ is a backward edge, i.e., $(v,u)$ is in the network, we update $f(v,u) \leftarrow f(v,u) - x(p)$;

The flow of other edges in the network (i.e., none of their corresponding forward edges or backward edges is in $p$) will not get affected. See an example in Figure 3.




$x(p) = 2$ $|f| = 7 + x(p) = 9$

Figure 3: Illustrating the procedure of augmenting continuing Figure 2. Suppose in the residual graph $G_f$ given in Figure 2 the algorithm identifies $s$-$t$ path $p = (s,a,c,t)$. Note that in this path $(s,a)$ and $(c,t)$ are forward edges and $(a,c)$ is backward edge. After augmenting $p$ the flow $f$ is given in the right figure, where affected flow are circled.

We emphasize that, after augmenting, $f$ remains a valid flow, i.e., $f$ satisfies both the capacity condition and the conservation condition. The capacity condition remains satisfied owes to how the residual graph

is constructed. Consider the two cases in augmenting: in either case, after augmenting, the flow remains non-negative and bounded by the capacity. Specifically,

1. if $e = (u,v) \in p$ is a forward edge, we know that $c_f(e) = c(u,v) - f(u,v)$ based on the construction of the residual graph. Since $x(p) \leq c_f(e)$, after augmenting the flow of edge $(u,v)$ becomes $f(u,v) + x(p) \leq f(u,v) + c_f(e) = f(u,v) + c(u,v) - f(u,v) = c(u,v)$. And it's obvious that $f(u,v) + x(p) \geq 0$.
2. if $e = (u,v) \in p$ is a backward edge, we know that $c_f(e) = f(v,u)$ based on the construction of the residual graph. Since $x(p) \leq c_f(e)$, after augmenting the flow of edge $(v,u)$ becomes $f(v,u) - x(p) \geq f(v,u) - c_f(e) = f(v,u) - f(v,u) = 0$. And it's obvious that $f(v,u) - x(p) \leq f(v,u) \leq c(v,u)$.

The reason why the conservation condition remains satisfied is that we augment an entire $s$-$t$ path. Hence, for any vertex $v$ in path $p$ except $s$ and $t$, the augmenting procedure adjusts the flow of two adjacent edges of $v$ in the network, and such adjustments always cancel out. For example, suppose that in path $p$ the two edges

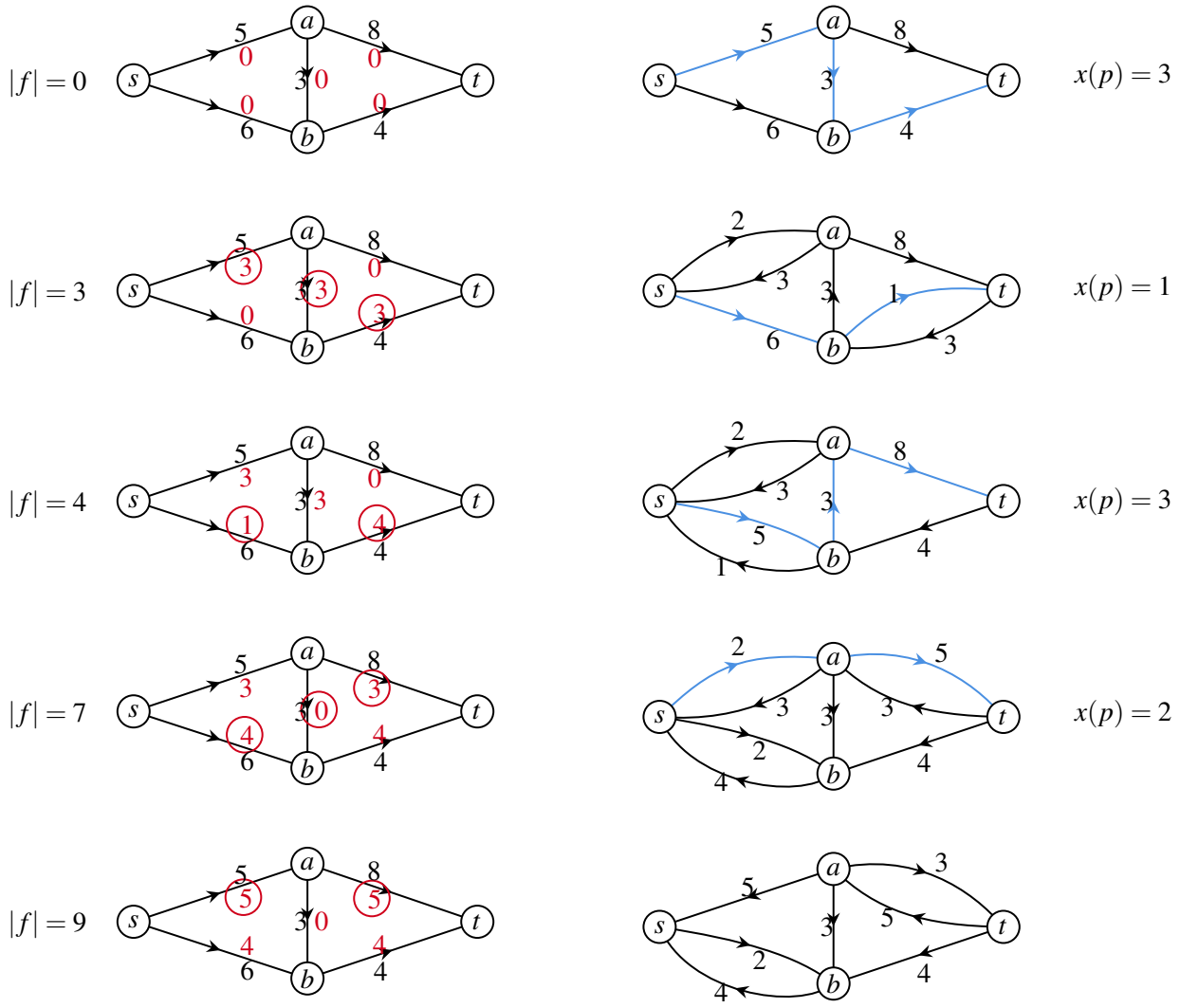



Figure 4: Running the FF algorithm on an instance. Each row shows an iteration. The left column shows the current flow $f$. The right column shows the residual graph w.r.t. $f$ and the $s$-$t$ path found (in blue).

are $(u,v)$ and $(v,w)$. If both edges are forward edges, then both $f(u,v)$ and $f(v,w)$ are increased by $x(p)$; so $v$ remains balanced; If $(u,v)$ is a forward edge and $(v,w)$ is a backward edge, then $f(u,v)$ is increased by $x(p)$ and $f(w,v)$ is decreased by $x(p)$; again, $v$ remains balanced; other two cases can be verified similarly.

After augmenting path $p$, the value of the new flow becomes $|f|+x(p)$. Since we assume that all edges in the residual graph has positive capacity, we have that $x(p)>0$, which means the value of the new flow always gets improved. See an example of running Ford-Fulkson on a small network, given in Figure 4.

The Ford-Fulkson algorithm runs in $O(|f^*|\cdot(|V|+|E|))$ time, where $f^*$ is the flow returned by the FF. This is because, each iteration (including building $G_f$, finding an $s$-$t$ path $p$ in $G_f$ with BFS/DFS, and augmenting $p$) takes $O(|V|+|E|)$ time, and in each iteration, the value of the flow is increased by at least 1 (assuming that all capacities are integers), so the number of iterations needed is $O(|f^*|)$.

## The Max-Flow Min-Cut Theorem

We now prove that the Ford-Fulkson algorithm is correct, i.e., when the algorithm terminates, the returned flow is a maximum-flow. Let $f^*$ be the flow when the FF terminates. Note that the residual graph $G_{f^*}$ w.r.t. $f^*$ does not have any $s$-$t$ path (or equivalently, $s$ cannot reach $t$ in $G_{f^*}$), because otherwise the FF algorithm will not terminate. Now we define a cut: let $S^*$ be the set of vertices in $G_{f^*}$ that can be reached from $s$, formally $S^* := \{v \in V \mid s \text{ can reach } v \text{ in } G_{f^*}\}$, and define $T^* = V \setminus S^*$. Clearly, $s \in S^*$. Since $s$ cannot reach $t$ in $G_{f^*}$, we know that $t \in T^*$. Hence $(S^*,T^*)$ is an $s$-$t$ cut. For example, in Figure 4, $S^* = \{s,b\}$ and $T^* = \{a,t\}$.

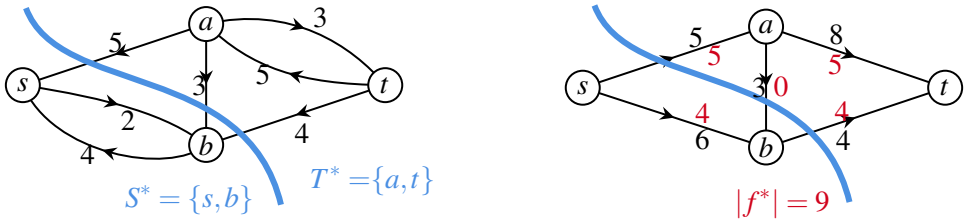



Figure 5: Illustration the proof of the max-flow min-cut theorem. Left: the residual graph $G_{f^*}$ wrt the flow $f^*$ when FF terminates (see Figure 4 of Lecture 19). The cut $(S^*,T^*)$ is constructed by collecting vertices reachable from $s$ in $G_{f^*}$ as $S^*$, and $T^* = V \setminus S^*$. Right: the same $s$-$t$ cut $(S^*,T^*)$ drawn on the (original) graph $G$, where one can see that for every edge $e \in E(S^*,T^*)$, $f^*(e) = c(e)$, and for every edge $e \in E(T^*,S^*)$, $f^*(e) = 0$.

We now show the central result: $|f^*| = c(S^*,T^*)$. Recall the Fact 2 in Lecture 18: the value of any flow, here we focus on $f^*$, can be calculated with any $s$-$t$ cut, and here we will use the particular cut $(S^*,T^*)$. Specifically, we can write $|f^*| = \sum_{e \in E(S^*,T^*)} f^*(e) - \sum_{e \in E(T^*,S^*)} f^*(e)$.

In fact, we have $f^*(e) = c(e)$ for every $e \in E(S^*,T^*)$. See Figure 5. Why? Suppose conversely that $f^*(e) < c(e)$. Assume that $e = (u,v)$; the fact that $e = (u,v) \in E(S^*,T^*)$ means that $u \in S^*$ and $v \in T^*$. According to the way the residual graph is constructed, in $G_{f^*}$, there will be a forward edge $(u,v)$. Consequently, $s$ can reach $v$ in $G_{f^*}$, as $s$ can reach $u$ (because $u \in S^*$) and $u$ can reach $v$ following this forward edge. This contradicts to the fact that $v \in T^*$.

Also, we can show that $f^*(e) = 0$ for every $e \in E(T^*,S^*)$. Why? Suppose conversely that $f^*(e) > 0$. Assume that $e = (u,v)$; the fact that $e = (u,v) \in E(T^*,S^*)$ means that $u \in T^*$ and $v \in S^*$. Then in the residual graph $G_{f^*}$, there will be a backward edge $(v,u)$. Consequently, $s$ can reach $u$ in $G_{f^*}$, as $s$ can reach $v$ (because $v \in S^*$) and $v$ can reach $u$ following this backward edge. This contradicts to the fact that $u \in T^*$.

Combining all above, we have $|f^*| = \sum_{e \in E(S^*,T^*)} f^*(e) - \sum_{e \in E(T^*,S^*)} f^*(e) = \sum_{e \in E(S^*,T^*)} c(e) - 0 = c(S^*,T^*)$.

This proves that $f^*$ is a maximum-flow and $(S^*,T^*)$ is a minimum *s*-*t* cut, i.e., the FF algorithm is optimal, and a minimum-cut can be constructed from the residual graph. It also completes the connection between the maximum-flow problem and the minimum-cut problem: for any network, the value of the maximum-flow always equal to the capacity of the minimum-cut. This is called the max-flow min-cut theorem.

# Lecture 33 Maximum-Matching and Minimum-Vertex Cover

## Maximum-Matching vs. Minimum-Vertex Cover on General Undirected Graphs

Let $G = (V, E)$ be an undirected graph. A *matching* $M \subset E$ is a subset of edges of $G$ that does not have common vertices. A matching is also called an *independent edge set*. See Figure 1 for an example. Given an undirected graph $G$, the so-called *maximum-matching problem* seeks a matching $M$ of $G$ such that the number of edges in the matching, i.e., $|M|$, is maximized.

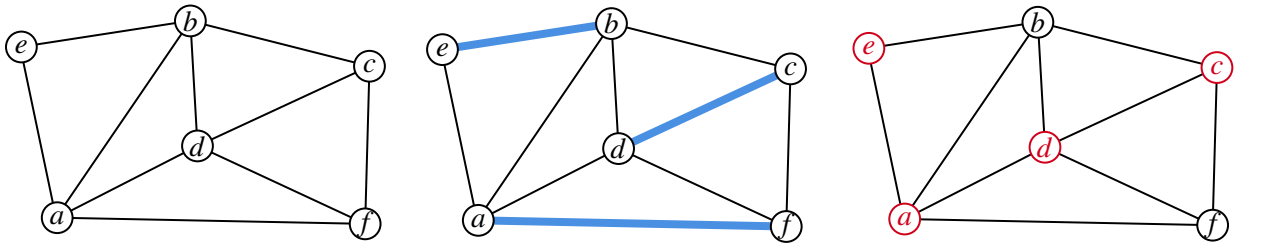



Figure 1: A matching $M$ and a vertex cover $V_1$ of an undirected graph $G$. Note that $M$ is also a *maximum matching* of $G$ and $V_1$ is also a minimum vertex cover of $G$.

A *vertex cover* of an undirected graph $G = (V, E)$ is defined as a subset of vertices $V_1 \subset V$ that *covers* all edges, i.e., for every edge $(u, v) \in E$ either $u \in V_1$ or $v \in V_1$ or both. Given an undirected graph $G$, the so-called *minimum-vertex cover problem* seeks a vertex cover $V_1$ of $G$ such that the number of verticesd in it, i.e., $|V_1|$, is minimized.

There exists a connection between matchings and vertex covers of an undirected graph, stated below.

**Claim 1.** Let $G = (V, E)$ be an undirected graph. Let $M$ be any matching of $G$ and let $V_1$ be any vertex cover of $G$. Then we have $|M| \leq |V_1|$.

*Proof.* Since $V_1$ is a vertex cover, it covers all edges of $G$, in particular, all edges of $M$. Since $M$ is a matching, all edges in $M$ does not share vertices, so it requires $|M|$ vertices to cover edges in $M$. Combined, we have $|M| \leq |V_1|$. □

So, the size of any vertex cover is an upper bound of the size of any matching. We again can virsualize such relationship with Figure 2. Clearly, if we can find a matching $M$ and a vertex cover $V_1$ (of the same undirected graph $G$) satisfying that $|M| = |V_1|$, then $M$ must be a maximum matching and $V_1$ must be a minimum vertex cover.

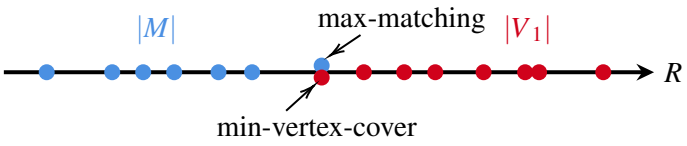



Figure 2: Each blue point represents the size of a matching and each red point represents the size of a vertex cover, of the same undirected graph. Do they always meet?

Does for *every* undirected graph there always exists a matching $M$ and a vertex cover $V_1$ that satisfies $|M| = |V_1|$? The answer is no, and Figure 1 gives such an example. A more informative example is a triangle: a graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$. In a triangle, the maximum matching

includes one edge, but to cover all 3 edges we need 2 vertices. In fact, an undirected graph consists of an *odd-length* cycle with $2m+1$ edges admits a maximum matching with $m$ edges and a minimum vertex cover with $m+1$ vertices. It seems that it's the odd-length cycles that results in a gap. In fact, this is true. Following we will show that, if an undirected graph $G$ does not contain odd-length cycle, then the size of its maximum matching must be equal to the size of its minimum vertex cover. The proof is again constructive: we will design an algorithm to find both (the algorithm is an application of max-flow), and show that they have equal size.

We give an equivalent (and nice) characterization of an undirected graph without odd-length cycles.

An *bipartite* graph, usually written as $G = (X \cup Y, E)$, is an undirected graph whose vertices can be separated into two disjoint subsets $X$ and $Y$ such that all its edges must span $X$ and $Y$ (i.e., every edge $e = (u,v) \in E$ must satisfy $u \in X$ and $v \in Y$). See Figure 3.

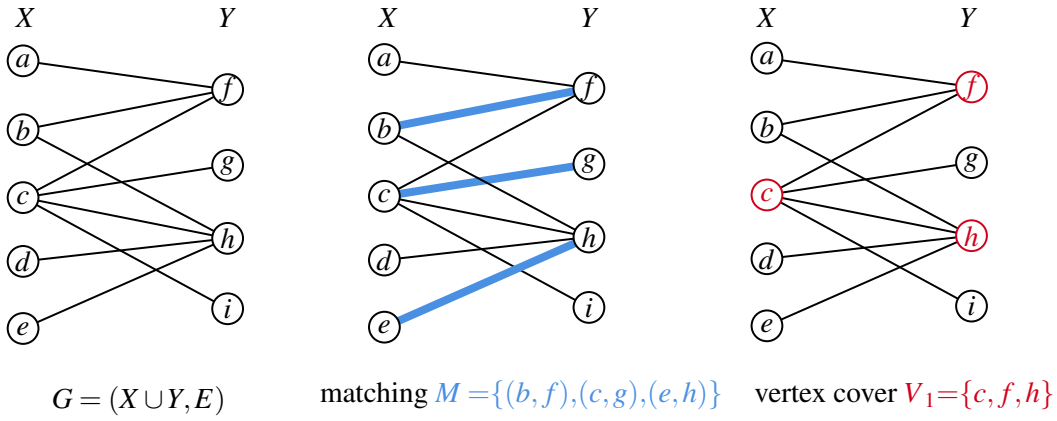



Figure 3: A bipartite graph $G = (X \cup Y, E)$, a maximum matching $M$, and a minimum vertex cover $V_1$ of $G$.

**Fact 1.** An undirected graph $G$ does not contain odd-length cycle if and only if $G$ is a *bipartite* graph.

*Proof.* We first prove that if $G$ is bipartite then any cycle in it must be even-length. (See cycle $(b,h,c,f)$ in Figure 3.) This is because any cycle need to visit $X$ and $Y$ alternately so its length must be even.

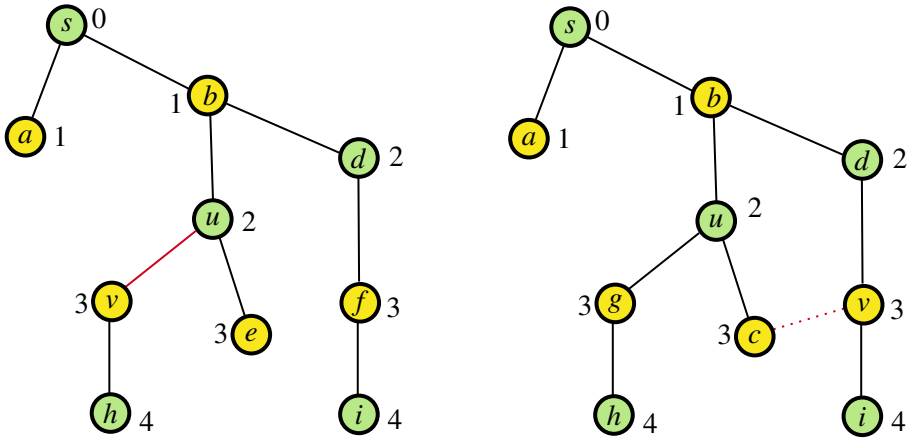



Figure 4: The shortest path tree $T$ where vertices are assigned to $X$ or $Y$ based on the parity of the distance from $s$. Left: an edge $(u,v) \in E$ is part of the tree. Right: an edge $(u,v) \in E$ is not part of the tree.

For the other side, assume that every cycle of undirected graph $G$ is even and we now prove that $G$ is bipartite. We assume that $G$ consists of one connected component; otherwise all connected components of $G$ can be processed separately in the same way below. Let $s$ be any vertex in $G$. Let $distance(s,v)$ as the length of the shortest path from $s$ to $v$. Let $T$ be the shortest path tree wrt $s$, i.e., the number of edges on the path from $s$ to $v$ is exactly $distance(s,v)$. See Figure 4. We now put $v \in V$ into $X$ if $distance(s,v)$ is even, and put $v \in C$ into $Y$ if $distance(s,v)$ is odd. Clearly, $X$ and $Y$ is a partition of all vertices. We now show that, any edge $(u,v) \in E$ must span $X$ and $Y$. There are two cases. The first one is that $(u,v)$ is also an edge in $T$. In this case we must have that $distance(s,v) = distance(s,u)+1$ so $u$ and $v$ must be assigned differently. (See Figure 4, Left.) The second case is that $(u,v) \notin T$. Suppose conversely that $distance(s,u)$ and $distance(s,v)$ have the same parity. Consider the $s$-$u$ path and the $s$-$v$ path in $T$. Let $b$ be the branching point of these two paths (also called the least common ancestor of $u$ and $v$). Then the $b$-$u$ path and the $b$-$v$ path are edge-disjoint and their lengths must also have the same parity. Now these two paths combined with edge $(u,v)$ must form a cycle of odd-length in $G$, a contradiction. (See Figure 4, Right.) □

## Maximum Matching and Minimum Vertex Cover of Bipartite Graphs

We design an algorithm to find one maximum matching $M$ of a given bipartite graph $G = (X \cup Y, E)$. This algorithm will also give a vertex cover $V_1$ of $G$, and we will prove $|M| = |V_1|$. This will therefore proves that $M$ is a maximum matching, $V_1$ is a minimum vertex cover, and that for every bipartite graph the size of its maximum matching and minimum vertex cover are equal.

We will not design new algorithm for this bipartite maximum matching problem, although this has been a classical problem in computer science and many algorithms exist (check wikipedia). To solve it, we will *transform this problem into a maximum-flow problem*. This typically consists of three steps:

1. construct an input for the maximum-flow problem, i.e., a network, based on the given input, i.e., a bipartite graph;
2. obtain the maximum flow and/or the minimum $s$-$t$ of the constructed network, using any existing algorithm such as Ford-Fulkson algorithm;
3. construct the maximum matching of the given bipartite graph based on the optimal solution in step 2.

Step 2 can be done simply by calling an existing solver. Steps 1 and 3 are closely related. How to construct the network is another "art" part of algorithm design: usually it requires trying and observing.

Here is a working approach. Given a bipartite graph $G = (X \cup Y, E)$, we construct a network $G' = (V', E')$ with source $s$ sink $t$ and edge capacities $c$ as follows. See Figure 5. We construct $G'$ by modifying $G$: not a surprise, the given bipartite graph is the main body of $G'$. We first add two new vertices, a source $s$ and a sink $t$, a step that is usually required in transforming a problem into a maximum-flow problem. That is, $V' = X \cup Y \cup \{s,t\}$. Second, we add a new directed edge $(s,x)$ that connects the source to every vertex $x \in X$, and add a new directed edge $(y,t)$ that connects every vertex $y \in Y$ to the sink $t$. For every edge $(x,y) \in E$ with $x \in X$ and $y \in Y$ in the bipartite graph, we then copy it to $G'$ and assign it with a direction pointing from $x$ to $y$. Last, we set the capacity as 1 for every edge in $G'$, i.e., $c(e) = 1$ for every $e \in E'$.

In step 2, we use any max-flow algorithm to find a maximum flow, denoted as $f^*$, of the constructed network $G'$. As all capacities of the network are integers, we can therefore assume that $f^*(e)$ is integer for any $e \in E'$. In particular, since all capacities are 1s, we know that $f^*(e) \in \{0,1\}$. Therefore, edges in the constructed network with $f^*(e) = 1$ forms a set of *edge-disjoint paths* from $s$ to $t$, and the number of such paths is $|f^*|$, as each such path carries 1 unit of flow. We collect those edges in the bipartite graph with $f^*(e) = 1$ as the

matching of the bipartite graph (although we have not proved them a matching yet). Formally, step 3 finds: $M := \{e \in E \mid f^*(e) = 1\}$.

We now prove that the identified edge-set $M$ is a maximum matching of the bipartite graph. First, we show $M$ is a matching, i.e., no two edges in $M$ share vertex. Suppose conversely that there exist two edges $(u,v),(w,v) \in M$ that share $v$. Then, by the construction of $M$, we know that $f^*(u,v) = f^(w,v) = 1$. Therefore, in the constructed network $G'$, the in-flow of $v$ is at least $f^*(u,v) + f^*(w,v) = 2$, so its out-flow must be at least 2. This constracts to that the *out-capacity* of $v$ is 1. From here we can also see why we set the capacities of the edges adjacent to $s$ or $t$ as 1: this guarantees that each vertex is matched at most once.

We then show that $M$ is a maximum matching of the bipartite graph $G$. First, notice that $|M| = |f^*|$. This is because $|f^*|$ equals to the number of above edge-disjoint $s$-$t$ paths, and we add the middle edge of each path to $M$. Therefore, if a matching $M'$ with more edges than $M$ exists, i.e., $|M'| > |M| = |f^*|$, we can use $M'$ to build $|M'|$ such edge-disjoint $s$-$t$ paths and thus give a flow with value of $|M'|$, which contradicts to the fact that $f^*$ is a maximum flow.

We continue above algorithm to find a (minimum) vertex cover of the bipartite graph $G$. The code, like the minimum $s$-$t$ cut, again hides in the residual graph $G'_{f^*}$ w.r.t. the maximum flow $f^*$. See Figure 6. Let $(S^*,T^*)$ be the minimum $s$-$t$ cut found using $G'_{f^*}$. Recall that $S^* := \{v \in V' \mid s \text{ can reach } v \text{ in } G'_{f^*}\}$, and

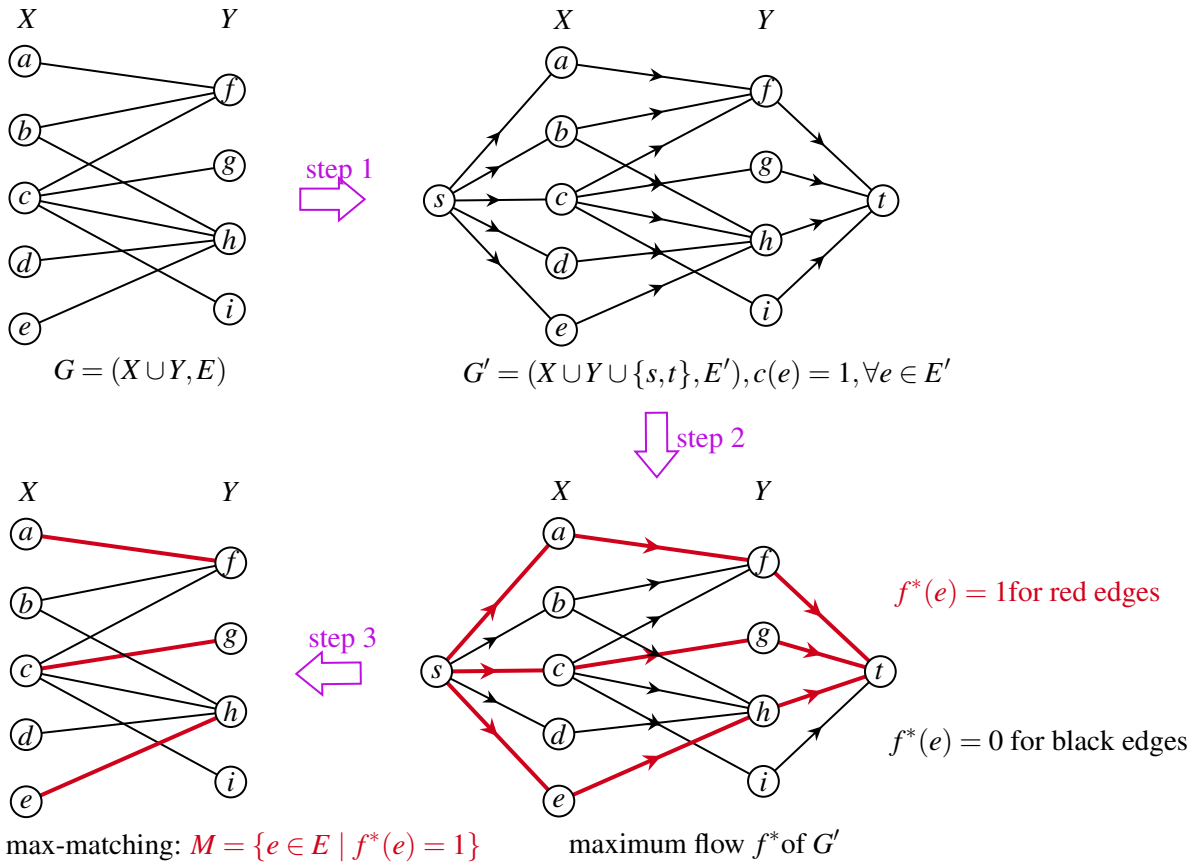



Figure 5: Solving the maximum-matching problem of bipartite graph using network flow. In the upper right network $G'$, every edge $e$ has a capacity of $c(e) = 1$. In the lower right figure which shows the maximum flow $f^*$, red edges indicate $f^*(e) = 1$. The maximum-matching of the bipartite graph $G$ is given by $M = \{e \in E \mid f^*(e) = 1\}$.

$T^* = V' \setminus S^*$. This $s$-$t$ cut partitions all vertices in the bipartite graph into 4 categories: $X \cap S^*$, $X \cap T^*$, $Y \cap S^*$, and $Y \cap T^*$. See the sketch of the network in the middle of Figure 6. We collect two of them: we define $V_1 := (X \cap T^*) \cup (Y \cap S^*)$ and we will prove that $V_1$ is a minimum vertex cover of the bipartite graph.

We first show that $V_1$ is a vertex cover of the bipartite graph. This is equivalent to show the following fact. Can you spot it in Figure 6?

**Fact 2.** In the bipartite graph, there does not exist any edge between $(X \cap S^*)$ and $(Y \cap T^*)$.

*Proof.* Suppose conversely that there exists an edge $(u,v) \in E$ with $u \in X \cap S^*$ and $v \in Y \cap T^*$. Consider $f^*(u,v)$, i.e., the amount of flow carried by this edge in the maximum flow $f^*$. There are two possibilities (as all capacities are 1): either $f^*(u,v) = 0$ or $f^*(u,v) = 1$. See Figure 7.

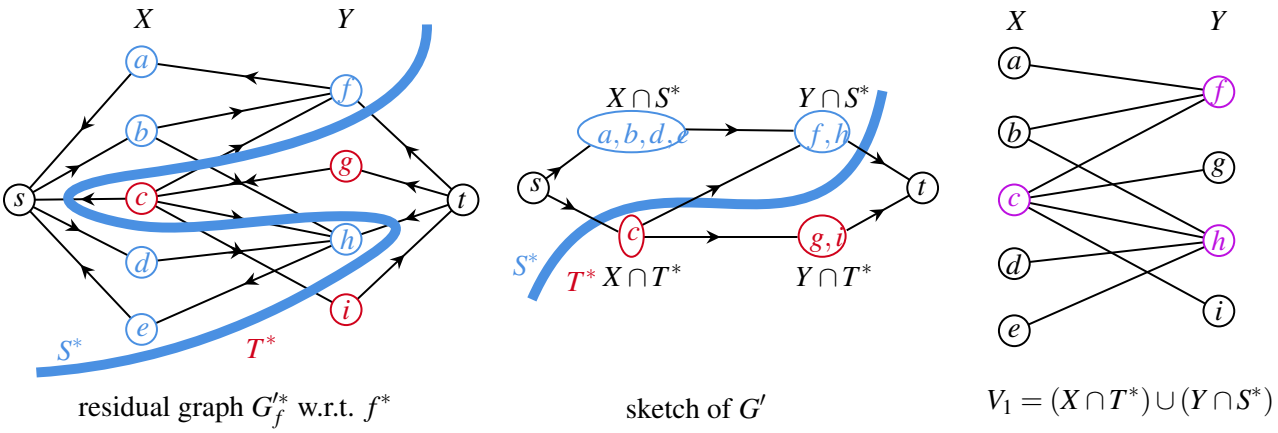
1. Suppose that $f^*(u,v) = 0$. Then in the residual graph $G'_{f^*}$ this edge becomes a forward edge $(u,v)$. Then $v$ must be in $S^*$ as $s$ can reach $u$ and $u$ can reach $v$ following this edge, a contradiction to the assumption that $v \in T^*$.

2. Suppose that $f^*(u,v) = 1$. Then we must have that $f^*(s,u) = f^*(v,t) = 1$, in order to make $u$ and $v$ satisfy the conservation condition. Therefore, in the residual graph $G'_{f^*}$, all these three edges become backward edges. Think: in the residual graph why $u$ is reachable from $s$? Clearly $u$ cannot be reached from $s$ using edge $(s,u)$ as it's edge $(u,s)$ that appears in the residual graph. Hence, there must exist a path from $s$ to $u$ using other intermediate vertices. Assume that on this path the vertex before $u$ is $w$, i.e., edge $(w,u)$ appears in the residual graph. Notice that $w$ cannot be $v$, as otherwise this implies that $s$ can reach $v$ which contradicts to that $v \in T^*$. Since edge $(w,u)$ is also a backward-edge, we have that $f^*(u,w) = 1$. Combined, we have a contradiction, as it's not possible to have both $f^*(u,v) = 1$ and $f^*(u,w) = 1$ because this breaks the capacity condition of edge $(s,u)$. □

Above fact immediately gives that $V_1$ is a vertex cover of the bipartite graph. We now show that $V_1$ is also a minimum vertex cover. Consider the capacity of $s$-$t$ cut $(S^*, T^*)$. See Figure 6 for a sketch of $G'$. Since there is no edge between $X \cap S^*$ and $Y \cap T^*$, we have that $c(S^*, T^*) = |X \cap T^*| + |Y \cap S^*|$, as every edge has a capacity of 1. According to the max-flow min-cut theorem, we know $c(S^*, T^*) = |f^*|$. We also proved that $|M| = |f^*|$. Combined, we have $|V_1| = |X \cap T^*| + |Y \cap S^*| = c(S^*, T^*) = |f^*| = |M|$, as desired.

We summarize the maximum matching problem and the minimum vertex cover problem below.

1. For any *bipartite graph*, its maximum matching and minimum vertex cover always have equal cardi-




residual graph $G'^*_f$ w.r.t. $f^*$ sketch of $G'$ $V_1 = (X \cap T^*) \cup (Y \cap S^*)$

Figure 6: Constructing the minimum vertex cover using the residual graph w.r.t. the maximum flow.

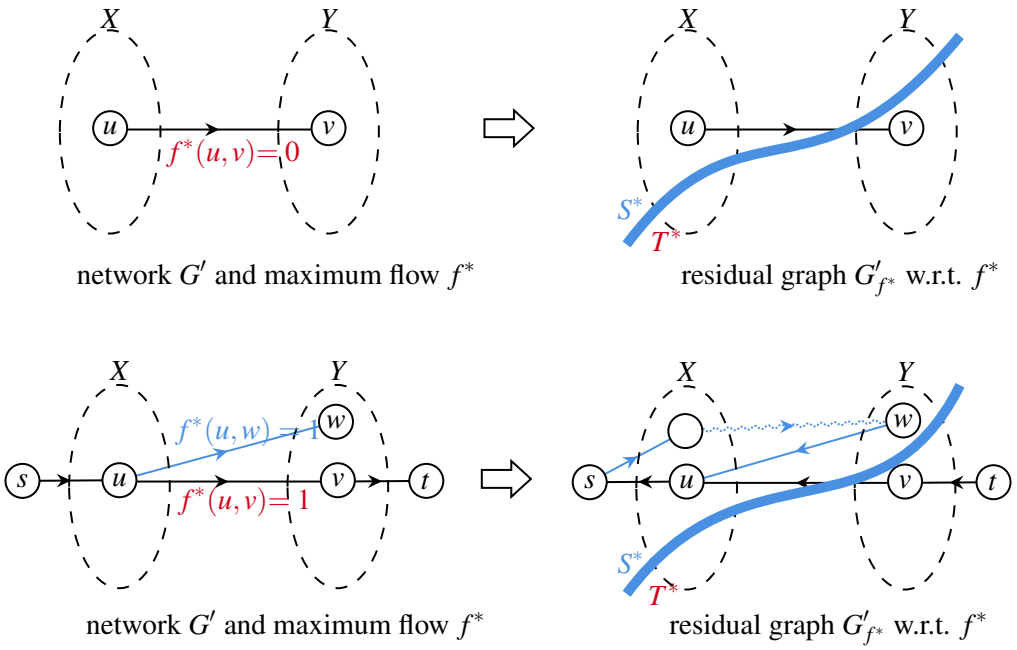



Figure 7: Illustrating the two cases in proving Fact 2.

nality, and such maximum matching and minimum vertex cover can be found using above algorithm which runs in polynomial-time.

2. For *general* undirected graphs, the size of any vertex cover is always an upper bound of that of any matching, but a gap might exist on graphs with odd-length cycles. The minimum vertex cover problem (on general undirected graphs) is NP-hard, i.e., there does not exist polynomial-time (optimal) algorithm for it. But the maximum matching problem (on general undirected graphs) can be solved in polynomial-time, for example, with the Blossom algorithm (Edmonds, 1961).

# Lecture 34 Project Selection Problem

Let $V$ be a set of projects, and there is a profit $p(v)$ associated with project $v \in V$. Note that the profit could be positive or negative. These projects are not independent: to accomplish a certain project, one need to accompolish all its *prerequisite* projects. Such dependency can be modeled as a directed graph $G = (V,E)$, where edge $(u,v) \in E$ represents that $v$ depends on $u$, i.e., $u$ is one of the prerequisite of $v$. We say a subset of projects $V_1 \subset V$ satisfies the *prerequisite condition* if for every $v \in V_1$, $(u,v) \in V_1$ implies that $u \in V_1$. Given a directed graph $G = (V,E)$ with profit $p(\cdot)$, the project selection problem seeks subset $V_1 \subset V$ such that $V_1$ satisfies the prerequisite condition and that the total profit of the selected projects i.e., $\sum_{v \in V_1} p(v)$ is maximized. See Figure 1 for an example.

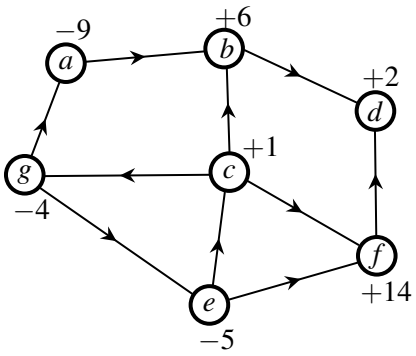



Figure 1: An example of project selection problem. $\{g,c,e\}$ satisfies the prerequisite condition; $\{b,c,d\}$ does not satisfies the prerequisite condition. The optimal solution for this example is $V_1 = \{g,c,e,f\}$, as $V_1$ satisfies the prerequisite condition and that its total profit $\sum_{v \in V_1} p(v) = 6$ is maximized.

The project selection problem is an abstracted formulation that well models application scenarios that involves making decisions over a set of things that depend on each other. For instance, this formulation can be used to model the election of courses. Specifically, each vertex represents a course and the prerequisites are modeled as edges. Each course is associated with an estimated gain/loss on the overall GPA. Hence, this formulation seeks a subset of courses (satisfying the prerequisite condition, of course) so as to maximize the net gain over one's GPA. Another application scanario is running a company, where one needs to decide over a set of "items" such as hiring people, purchasing equipments, making products, etc. These items clearly depend on each other. For examples, making a product requires hiring people with certain expertise and purchasing certain equipments; a certain equipment can be used by several products, etc. The items can be modeled as vertices and their dependencies be modeled as edges. Items may have an estimated cost (negative profit) or gain (positive profit). Deciding on the items so as to maximize the total profit is clearly another instance of the project selection problem.

We design algorithm to solve this problem. The brute-force approach enumerates all possible subsets (which projects to select), and for each subset one can examine if it satisfies the prerequisite condition, and if it does, calculate its total profit and keep track of the one with maximized total profit. Since there are $2^{|V|}$ possible subsets, this brute-force algorithm certainly runs in exponential time.

We now design a polynomial-time algorithm for above problem, by reducing/transforming it into a network flow problem. Recall that, such reduction involves a 3-step procedure. First, constructing a network $(G',s,t,c)$ based on the input graph $G$ and profit $p(\cdot)$; second, using an existing algorithm for max-flow to obtain the maximum-flow $f^*$ and/or minimum $s$-$t$ cut $(S^*,T^*)$ of the network $G'$; third, find the optimal solution $V_1$, using the found $f^*$ and/or $(S^*,T^*)$.

You may pause now to think about how we apply this framework to solve the maximum-matching problem for bipartite graph. For that problem, we establish that, there exists a one-to-one correspondence between any matching $M$ of the bipartite graph and any (integral) flow $f$ of the constructed network, and that $|M| = |f|$. Hence, finding maximum-matching of the bipartite graph is equivalent to finding maximum-flow of the network. But it is hard to find any connection between the project selection problem with flow. In fact, here we essentially seek a partition of vertices—projects we select and projects we do not select, which is exactly a cut! Hence, for the project selection problem, we instead will establish a one-to-one correspondence between any selection and any $s$-$t$ cut of the constructed network, and we will also show that minimizing the total profit of the selected projects is equivalent to finding the minimum $s$-$t$ cut of the network.

We now work out the details of above roadmap. In step #1, we construct the network $(G' = (V', E'), s, t, c)$ as follows. See Figure 2. The directed graph $G'$ incorporates the given graph $G = (V, E)$, not a surprise. We add a source $s$ and a sink $t$, i.e., $V' = V \cup \{s, t\}$. We connect $s$ to each project with negative profit, and connect each project with positive profit to sink $t$. Formally, edges in the network is: $E' = E \cup \{(s, v) \mid p(v) < 0\} \cup \{(v, t) \mid p(v) > 0\}$. Now we assign capacities: edges in $E$, i.e., these in the given graph $G$, will have a capacity of $\infty$; edge $(s, v)$ will have a capacity of $-p(v)$; edge $(v, t)$ will have a capacity of $p(v)$. The reasons behind such construction will be revealed when proving the correctness of the algorithm.

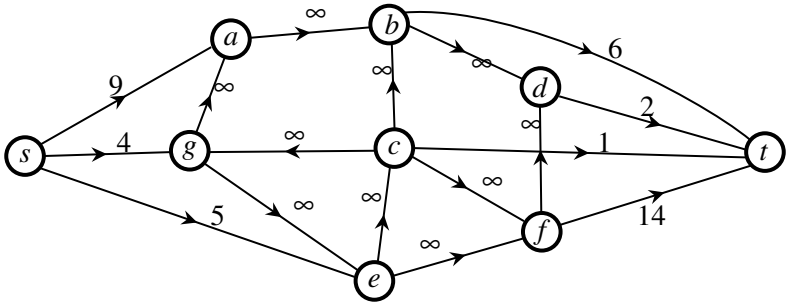



Figure 2: The constructed network $(G', s, t, c)$ for the example in Figure 1.

In step #2, we find a minimum $s$-$t$ cut $(S^*, T^*)$ of above network $(G', s, t, c)$. In step #3, the algorithm simply returns projects in $t$-side, i.e., returning $V_1^* = T^* \setminus \{t\}$. See Figure 3.

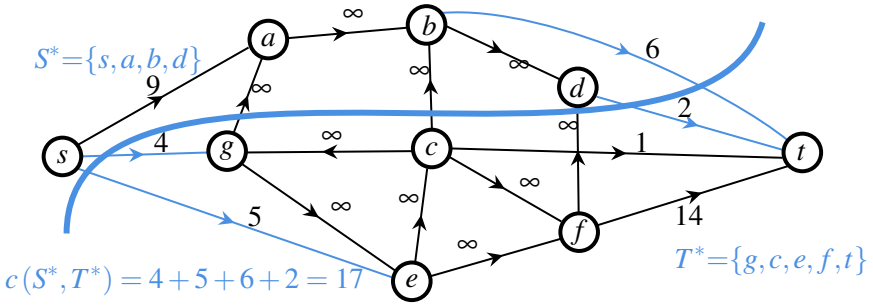



Figure 3: The minimum $s$-$t$ cut $(S^*, T^*)$ of the network $G'$ in Figure 2. Therefore, for this example, the algorithm returns $V_1^* = T^* \setminus \{t\} = \{g, c, e, f\}$.

We now prove that the above algorithm is correct, i.e., to prove that the returned $V_1^*$ satisfies the prerequisite condition and that $\sum_{v \in V_1^*} p(v)$ is maximized. To achieve this, we establish the one-to-one correspondence between selections and $s$-$t$ cuts of the built network. Formally, let $V_1 \subset V$ be an arbitrary selection (i.e., a subset of projects). Let $T = V_1 \cup \{t\}$ and $S = (V \setminus V_1) \cup \{s\}$. Clearly, $(S, T)$ forms an $s$-$t$ cut of the network.

**Fact 1.** $V_1$ satisfies the prerequisite condition if and only if $c(S,T) < \infty$.

You may check a couple of examples to fully understand above statement. See Figure 4.

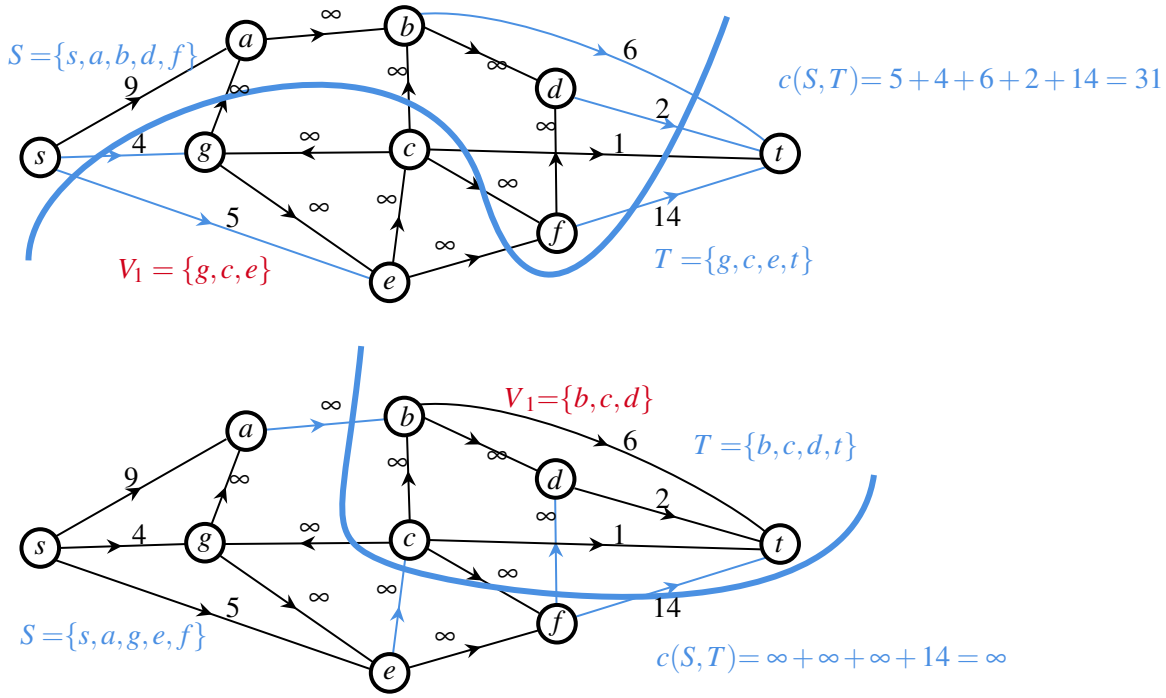



Figure 4: Above: $V_1 = \{g,c,e\}$ which satisfies the prerequisite condition; the corresponding cut capacity $c(S,T) = 31 < \infty$. Below: $V_1 = \{b,c,d\}$ which does not satisfy the prerequisite condition; the corresponding cut capacity $c(S,T) = \infty$. In both examples, cut-edges are marked in blue.

*Proof of Fact 1.* If $V_1$ satisfies the prerequisite condition (see the top example of Figure 4), then, by definition, there is no such edge $(u,v)$ where $v \in V_1$ but $u \notin V_1$. Hence, none of edges in $E$, which have capacity of $\infty$, belongs to the cut-edges of cut $(S,T)$. So we must have $c(S,T) < \infty$. If $V_1$ does not satisfy the prerequisite condition (see the bottom example in Figure 4), then there exists edge $(u,v)$ where $v \in V_1$ and $u \notin V_1$. This edge is therefore one of the cut-edges of cut $(S,T)$. Also because the capacity of this edge is $\infty$, we must have $c(S,T) = \infty$. □

In the network $G'$, $s$-$t$ cut $(S,T)$ with $c(S,T) < \infty$ exists; for example the $s$-$t$ cut with just $\{s\}$ on one side. Hence, the minimum $s$-$t$ cut $(S^*,T^*)$ of the network must have that $c(S^*,T^*) < \infty$, since it is minimum. Combining with above Fact 1, we have the following:

**Fact 2.** The returned $V_1^* = T^* \setminus \{t\}$ satisfies the prerequisite condition.

It should be clear now why we assign infinite capacity to edges in $E$ when constructing the network $G'$. This is simply because doing so guarantees that the returned projects satisfies the prerequisite condition. In other words, assigning these edges an infinite capacity *avoids* the cut-edges go through any of them, ensuring that the projects on the $t$-side satisfies the prerequisite condition.

We now prove that $V_1^*$ maximizes the total profit, by establishing the quantitative relationship between the total profit of any selection and the capacity of the corresponding $s$-$t$ cut. Let $V_1 \subset V$ be a subset of $V$ that satisfies the prerequisite condition. Let $(S,T)$ be the corresponding $s$-$t$ cut, where $T = V_1 \cup \{t\}$ and let $S = (V \setminus V_1) \cup \{s\}$. $S$ and $T$ partition all projects (vertices in $V$) into two classes. Combining whether a

project has a negative or positive profit, all projects are now partitioned into four categories: $S_+$, $S_-$, $T_+$, and $T_-$, where $S_+$ consists of projects in $S$ while having positive profit (similar for other three categories). Now the network $G'$ can be sketched in Figure 5. Note that $s$ connects to projects with negative profit, i.e., $S_- \cup T_-$, and $t$ is connected from projects with positive profit, i.e., $S_+ \cup T_+$. Also note that there is no edge from $S_- \cup S_+$ to $T_- \cup T_+$ because we assume that $V_1$, which is $T_- \cup T_+$, satisfies the prerequisite condition (see Fact 1).

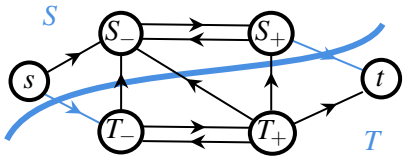



Figure 5: Sketch of the network $G'$ for an $s$-$t$ cut $(S,T)$.

Now let's establish the connection between capacity of $(S,T)$ and profit of $V_1$. Note that the cut-edges of $(S,T)$ consists of edges from $s$ to $T_-$ and edges from $S_+$ to $t$. We have: $c(S,T) = \sum_{v \in T_-} -p(v) + \sum_{v \in S_+} p(v)$. The total profit of $V_1$ is: $\sum_{v \in V_1} p(v) = \sum_{v \in T_-} p(v) + \sum_{v \in T_+} p(v)$. We sum up both sides. Note that term $\sum_{v \in T_-} p(v)$ cancels out. We have $c(S,T) + \sum_{v \in V_1} p(v) = \sum_{v \in S_+} p(v) + \sum_{v \in T_+} p(v)$. Note that the right side is exactly the total profit of projects with positive profit, i.e., $\sum_{v \in V : p(v) > 0} p(v)$, which is independent of the $V_1$ or $(S,T)$. In other words, $c(S,T) + \sum_{v \in V_1} p(v)$ is a constant! Hence, minimizing one term is equivalent to maximizing the other. Since the algorithm finds $(S^*, T^*)$ which minimizes $c(S,T)$, the resulting $V_1^*$ must have the maximized total profit. This is summarized below, which also concludes the proof.

**Fact 3.** The returned $V_1^* = T^* \setminus \{t\}$ maximizes the total profit.

The time complexity of the algorithm (i.e., the 3 steps) is dominated by the time complexity of calling a max-flow solver. Note that the Ford-Fulkson algorithm does not run in polynomial-time (it in facts runs in pseudo-polynomial-time). But there exists polynomial-time algorithm for maximum-flow, for example the Push-Relabel algorithm. Hence, the algorithm for project selection problem can be solved in polynomial-time as well.

# Lecture 35 Polynomial-Time Reduction

In Lecture 39, we designed an algorithm to solve the maximum matching problem on bipartite graphs by *transforming* it into a maximum-flow problem. This algorithm calls once a solver for the maximum-flow problem (in step 2), and it also builds the network (in step 1) and fetch the matching by processing the maximum-flow (in step 3). This is a typical example of so-called *polynomial-time reduction*. Such reduction is extremely useful—not only in solving new problems by using existing algorithms, but also in proving that a problem is hard. Below we first formally define it.

**Definition 1** (Polynomial-Time Reduction)**.** Let $X$ and $Y$ be two problems. If there exists an algorithm for $X$ which calls a polynomial number of an algorithm for $Y$ and uses polynomial number of extra computational steps, then we say $X$ is polynomial-time reducible to $Y$, denoted as $X \leq_p Y$.

In short, problem $X$ is polynomial-time reducible to problem $Y$ means "we can design an algorithm for $X$ using an algorithm for $Y$". Of course, in this algorithm for $X$ the number of calls of a solver for $Y$ is limited to polynomial, and this algorithm for $X$ can have extra procedures, but these again need to run in polynomial-time.

Our algorithm for solving the maximum matching problem on bipartite graphs is such an example: in that algorithm, we called once a solver for the maximum-flow problem; and the steps 1 and 3 takes polynomial-time. Therefore, we have (maximum-matching-problem-on-bipartite-graph) $\leq_p$ (maximum-flow-problem).

Note that, above definition of polynomial-time reduction does *not* require that the algorithm for $Y$ runs in polynomial-time. It does not matter whether or not $Y$ can be solved in polynomial-time. But, *if* the solver for $Y$ runs in polynomial-time, then the resulting algorithm for $X$ runs in polynomial-time as well, because a polynomial number of calls of an algorithm runs in polynomial-time also takes polynomial-time. Formally,

**Fact 1.** If $X \leq_p Y$ and $Y$ can be solved in polynomial-time, then $X$ can be solved in polynomial-time.

## Decision Problems

A *decision problem* refers to a problem with a *binary* output, i.e., an algorithm for this problem answers "yes" or "no". For example, the 3SAT problem, which asks if a set of given clauses can be satisfied, is a decision problem. Many of the problems we have seen in this course are *optimization problems*. For examples, the maximum-flow problems asks a flow, which is a function that assigns a number to each edge, of a given network, such that its value is maximized; the (minimum) vertex cover problem asks a vertex cover, which is a subset of vertices, of a given undirected graph, such that its cardinality is minimized. These optimization problems are not decision problems.

Usually we can define a *decision-version* of an optimization problem. Let's try it for the vertex cover problem; its decision-version can be defined as follows: given an undirected graph $G = (V, E)$ and an integer $k \geq 0$, decide if there exists a vertex cover $V_1$ of $G$ such that $|V_1| \leq k$. In other words, the problem asks if the given $k$ is an upper-bound of the cardinality of the minimum vertex cover of the given undirected graph.

We use $VC(G)$ to refer to the (optimization-version) of the vertex-cover problem, and use $VC(G,k)$ to refer to above decision-version of the vertex-cover problem. What's the ralationship between them? Here by "relationship" we specifically ask if one problem is polynomial-time reducible to the other. Intuitively, the decision-version is "easier" than the optimization-version, and we can easily use an algorithm for the optimization-version to solver the decision-version.

**Fact 2.** $VC(G,k) \leq_p VC(G)$.

*Proof.* To prove that $VC(G,k)$ is polynomial-time reducible to $VC(G)$, by definition, we need to design an

algorithm for $VC(G,k)$, in which we can assume and use, up to polynomial number of calls, a solver for $VC(G)$. Here is such an algorithm:

```
algorithm-for-VC(G,k)
    call the solver for VC(G) to get the minimum vertex cover of G: V*_1 ← solver-for-VC(G);
    if |V*_1| ≤ k: return "yes";
    else: return "no";
end;
```

Clearly, this algorithm for $VC(G,k)$ is correct; it calls once (and hence polynomial number) a solver for $VC(G)$, and the additional procedure (which just compares $|V_1^*|$ with the given $k$) runs in constant time (and hence in polynomial-time). This proves that $VC(G,k) \leq_p VC(G)$. □

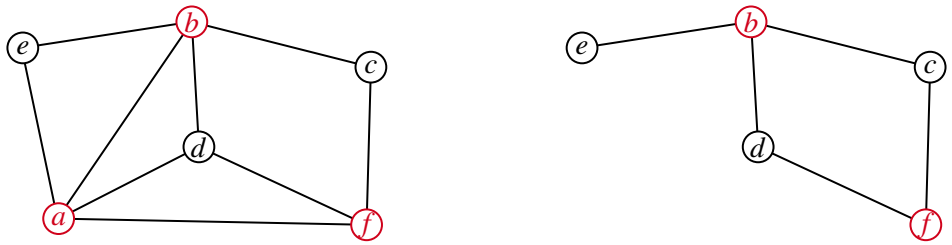
How about the other side? Is $VC(G)$ polynomial-time reducible to $VC(G,k)$? It's not obvious but this is true as well. Formally,

**Fact 3.** $VC(G) \leq_p VC(G,k)$.

*Proof.* Again we need to design an algorithm for $VC(G)$ using a solver for $VC(G,k)$. Recall that a solver for the decision-version tells if a given $k$ is an upper-bound of the cardinality of the minimum vertex cover of $G$. Therefore we can calculate the *cardinality* of the minimum vertex cover of $G$, by testing all possible sizes of a subset of vertices in *increasing* order, i.e., $0,1,2,\cdots,|V|$; the first achievable value is the *cardinality* of the minimum vertex cover.

```
algorithm-for-cardinality-VC(G)
    for k = 0 → |V|
        call the solver for the decision-version to test the given G and this k: B ← solver-for-VC(G,k);
        if B = "yes": return k;
    end for;
end;
```

This algorithm finds the cardinality of the minimum vertex cover of $G$; we denote it by $k^*$. How can we then find the actual vertex cover? Here is one strategy: we determine a single vertex that must be in one minimum vertex cover; if this can be done, we can then include that vertex (to the minimum vertex cover) and resolve the "remaining part" of the graph recursively. The core procedure is therefore to determine if a vertex is in one minimum vertex cover, which can be solved based on this fact: let $v \in V$ be a vertex; there exists a minimum vertex cover that includes $v$ if and only if the "remaining-graph" $G_{-v}$, defined as the graph by removing $v$ and adjacent edges of $v$ from $G$, can be covered by $(k^*-1)$ vertices. See Figure 1. Note that, the solver for the decision-version can be used to answer whether any graph, in this case the remaining graph $G_{-v}$, can be covered by $(k^*-1)$ vertices.




$G = (V, E)$
one minimum vertex cover = $\{a, b, f\}$

$G_{-a} = (V \setminus \{a\}, E')$
one minimum vertex cover = $\{b, f\}$

Figure 1: Illustrating that a vertex $v$ is in one minimum vertex cover if and only if $G_{-v}$ can be covered by $k^* - 1$ vertices, where $k^*$ is defined as the cardinality of any minimum vertex cover of $G$.

```
algorithm-for-VC(G)
    k* ← algorithm-for-cardinality-VC(G);
    for each v ∈ V
        build the remaining graph G_{-v} by removing v and its adjacent edges from G;
        call the solver for the decision-version to test G_{-v} and k* − 1: B ← solver-for-VC(G_{-v}, k* − 1);
        if B = "yes": save this v as u and break the for-loop;
    end for;
    resolve the remaining graph G_{-u}: V_2* ← algorithm-for-VC(G_{-u});
    return {u} ∪ V_2*;
end;
```

The complete algorithm for the optimization version is given above. Note that this algorithm is a recursive algorithm, and we define that algorithm-for-$VC(G)$ returns the actual minimum vertex cover of $G$. What's its running time? Note that it calls itself once, i.e., algorithm-for-$VC(G_{-u})$, and the number of vertices in this call is reduced by 1. Besides, it calls algorithm-for-cardinality-$VC(G)$, which requires $|V|$ calls to solver-for-$VC(G,k)$, and the for-loop requires another $|V|$ calls to solver-for-$VC(G,k)$. Combined, its running time $T(n)$, where $n = |V|$, can be written as $T(n) = T(n-1) + 2n \cdot solver(G,k)$, where $solver(G,k)$ represents the running time of calling solver-for-$VC(G,k)$ once. Solving this recurrence gives $T(n) = O(n^2) \cdot solver(G,k)$.

This proves $VC(G) \leq_p VC(G,k)$, as we designed an algorithm for $VC(G)$, which uses $O(n^2)$, a polynomial number, of calls to a solver for $VC(G,k)$. □

These two facts show that the decision-version and the optimization-version of the vertex cover problem are essentially equivalent, in terms of that they are mutually polynomial-time reducible. Such conclusion are true for most optimization problems. This is useful in simplying the hierarchy of problems: now we can solely focus on decision problems, as they are simpler (the output is just a single bit), while also captures the hardness of the optimization version (equivalent in terms of polynomial-time reduction).

## P and NP

In complexity theory, "P" is defined as the set of decision problems that are *polynomial-time solvable*, and "NP" is defined as the set of decision problems that are *polynomial-time verifiable*, Polynomial-time solvable means that one can design a polynomial-time algorithm to *solve* the problem. Polynomial-time verifiable means that one can design a polynomial-time algorithm to *verify* if a *given* solution (also called a *certificate*)

is correct, and such certificate must exist for "yes"-instances. They are formally defined below.

**Definition 2** (P)**.** Let $X$ be a decision problem. We say $X \in$ P if there exists a polynomial-time algorithm $A$ (we call $A$ as a *solver*), such that,
(1), solver $A$ takes an instance $x$ of $X$ as input, and
(2), for every instance $x$ of problem $X$:
if the ground-truth of $x$ is "yes", then $A(x)$ returns "yes";
if the ground-truth of $x$ is "no", then $A(x)$ returns "no".

In above definition, condition (1) means that algorithm $A$ is *for solving* problem $X$; condition (2) means that algorithm $A$ is correct: every instance has a "ground-truth" answer, and algorithm $A$ always finds it.

**Definition 3** (NP)**.** Let $X$ be a decision problem. We say $X \in$ NP if there exists a polynomial-time algorithm $A$ (we call $A$ as a *verifier*), such that,
(1), verifier $A$ takes an instance $x$ of $X$ *and* a polynomial-length certificate $c$ as input, and
(2), for every instance $x$ of problem $X$:
if the ground-truth of $x$ is "yes", then there must exist a certificate $c$ such that $A(x,c)$ returns "yes";
if the ground-truth of $x$ is "no", then for any polynomial-length certificate $c$, $A(x,c)$ always returns "no".

In above definition, condition (1) means that algorithm $A$ is for verifying if the (given) certificate is the correct solution of the (given) instance; condition (2) means there exists (polynomial-length) certificate for every "yes"-instnace but not for any "no"-instance, and they can be correctly verified by algorithm $A$. Notice that, *how to find* such correct certificate is not the job of algorithm $A$. When we use this definition to show that a problem is in NP, we just need to argue that such certificate *exists*, we don't need to design algorithms to find it out. We give two examples below.

**Fact 4.** 3SAT problem is in NP.

*Proof.* By definition, to prove 3SAT is in NP, we need to design an algorithm (i.e., the verifier $A$), and show that it satisfies the two conditions. For 3SAT, the certificate is an assignment for all variables, which can be represented as a binary vector. The verifier, given below, will check if the given binary vector is in correct length, and if it satisfies all clauses, and these can be done in polynomial-time.

```
verifier (x, c)
    verify that the length of c equals to the number of variables in instance x: if not, return "no";
    verify that every clause in instance x is true when x_i is assigned with c[i]: if not, return "no";
    return "yes";
end;
```

Now we prove that above verifier satisfies the two conditions. It clearly takes an instance and a certificate as input. If $x$ is an "yes"-instance, then it means $x$ is satisfiable, i.e., there exists an assignment that satisfies all clauses. Clearly, once this assignment is passed to the verifier together with $x$, the verifier will return "yes". If $x$ is an "no"-instance, then it means $x$ is not satisfiable, i.e., there does not exist an assignment that satisfies all clauses. Hence, no matter which assignment is passed to the verifier with $x$, the verifier always returns "no". Also, verifier $A$ runs in polynomial-time. This proves that 3SAT is in NP. □

**Fact 5.** The decision-version of vertex cover problem, i.e., $VC(G,k)$, is in NP.

*Proof.* Again we design an algorithm (i.e., the verifier $A$), and show that it satisfies the two conditions. For $VC(G,k)$, the certificate is a subset of vertices, which can be represented as a vector. The verifier, given below, will verify its size, and verify if it covers all edges.

verifier $(x = (G,k), c)$
    verify the size of $c$ is at most $k$: if not, return "no";
    verify the vertices in $c$ can cover all edges of $G$: if not, return "no";
    return "yes";
end;

Above verifier takes an instance and a certificate as input. If $x = (G,k)$ is an "yes"-instance, then it means there exists a subset of vertices $V_1$ with $|V_1| \leq k$ and $V_1$ covers all edges in $G$. Hence, once this $V_1$ is passed to the verifier together with $x$, the verifier will return "yes". If $x$ is an "no"-instance, then it does not exist a subset with at most $k$ vertices that covers all edges in $G$. Hence, no matter which subset of vertices (coded in $c$) is passed to the verifier with $x$, the verifier always returns "no"—either because it exceeds $k$ vertices, or does not cover all edges. Also, verifier $A$ runs in polynomial-time. □

# Lecture 36 NP-complete and NP-hard

## Polynomial-Time Reduction Revisit

Polynomial-time reduction can be used in two different ways. Suppose we have $X \leq_p Y$.

1. If we know an algorithm to solve $Y$, then we know how to solve $X$. In this way we are making use of an (existing) algorithm for $Y$ to solve a (new) problem $X$. Such an example is we use the maximum flow problem (problem $Y$) to solve the bipartite maximum matching problem (problem $X$).

2. If we know problem $X$ is *hard*, then we know that problem $Y$ is *hard*. This is because $X \leq_p Y$ means that $Y$ is at least as hard as $X$, as any algorithm of $Y$ can be used to solve $X$. In this way, we are making use of an (existing) hard problem $X$ to prove that a (new) problem $Y$ is hard. Below we will formally define "hard". Informally, it means a polynomial-time algorithm does not exist (unless P = NP).

## NP-complete

**Definition 1** (NP-complete)**.** Let $X$ be a decision problem. We say $X$ is NP-complete, if $X$ is in NP, and for every problem $X'$ in NP, we have $X' \leq_p X$.

NP-complete problems are the "hardest" problems in NP. It's not obvious that NP-complete problems even exist. Thanks to the pioneers of computer science: in 1971 Cook and Levin proved the first NP-complete problem by showing that the circuit satisfiability problem is NP-complete. This builds the foundation of complexity theory.

One side of application of NP-complete relates to the connection between P and NP. Clearly, P is a subset of NP, i.e., P $\subset$ NP. But is the case that P $\neq$ NP (i.e., P is a strict subset of NP), or P = NP? This question remains open, and is perhap *the* most important open question in computer science. (It is also one of the seven Millennium Prize Problems selected by the Clay Mathematics Institute.) The existence of NP-complete problems gives a way to prove that P = NP, by just showing one NP-complete problem is polynomial-time solvable. This is easy to understand: NP-complete problems are the hardest problems in NP; if one of them is polynomial-time solvable, then every problem must be polynomial-time solvable. We formally state it and prove it below.

**Claim 1.** Let $X$ be a NP-complete problem. If $X$ is in P, then we have P = NP.

*Proof.* In order to show P = NP, we only need to prove NP $\subset$ P, as we already have P $\subset$ NP. Let $X'$ be an arbitrary problem in NP. Since $X$ is NP-complete, by definition, we know that $X' \leq_p X$. Because $X$ is in P, then we have that $X'$ is in P as well (see Fact 1 in Lecture 40). This proves that NP $\subset$ P, as $X'$ is arbitrary. □

Above claims suggests the two possible cases between P, NP, and NPC—here we use "NPC" to represent the set of all NP-complete problems. See Figure 1.

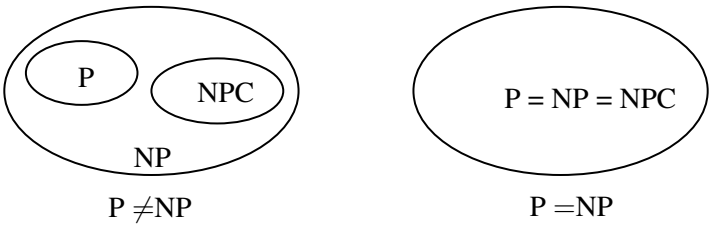



Figure 1: Two possibilities among P, NP, and NPC.

1. P $\neq$ NP. In this case, we must have P $\cap$ NPC $= \emptyset$. This is because otherwise, if there exists a problem $X \in$ P $\cap$ NPC, P will be equal to NP according to above Claim).

2. P $=$ NP. In this case, we must have P $=$ NP $=$ NPC. This is because in this case *any* two problems $X$ and $Y$ in NP (and therefore in P) must satisfy that $X \leq_p Y$ and $Y \leq_p X$, as both $X$ and $Y$ can be solved in polynomial-time. This proves that every problem in NP is NP-complete.

Another side of application of NP-complete problems is to use existint NP-complete problems to prove new problems are NP-complete (i.e., hard). This is based on the following claim.

**Claim 2.** Let $X$ be a problem in NP. If there exists an NP-complete problem $Y$ such that $Y \leq_p X$, we we have $X$ is NP-complete.

This is easy to understand: $Y$ is NP-complete, i.e., it is (one of) the hardest problem in NP; of course $Y$ is at least as hard as $X$ because $X$ is in NP. Now we have $Y \leq_p X$, i.e., $X$ is at least as hard as $Y$. Combined, $X$ is (one of) the hardest problem in NP as well. Below we give a formal proof.

*Proof.* Let $X'$ an arbitrary problem in NP. Since problem $Y$ is NP-complete, by definition, we have $X' \leq_p Y$. Since $Y \leq_p X$, we must have that $X' \leq X$. (Here we use the property that the polynomial-time reduction is transitive.) Combining that $X$ is in NP, we know that $X$ is NP-complete. □

Above Claim suggest a procedure to prove that a decision problem $X$ is NP-complete.

1. show that $X$ is in NP;

2. pick an existing NP-complete problem $Y$;

3. show that $Y$ is polynomial-time reducible to $X$.

Using above procedure, lots of problems have been proved to be NP-complete. These includes the 3SAT problem, the decision-version of the vertex cover problem, i.e., $VC(G,k)$. Other well-known NP-complete problems include, the decision-version of independenet set problem, the Hamiltonian path problem, the Hamiltonian cycle problem, 3D matching problem, $k$-coloring problem when $k \geq 3$, the subset-sum problem, decision-version of TSP (traveling salesman problems), etc. Below we give some details for the independent set problem.

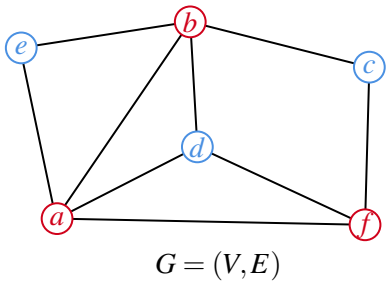
## The Independent Set Problem

An *independent set* of an undirected graph $G = (V,E)$ is a subset of vertices $V_1 \subset V$ satisfying that no two vertices in it is connected by an edge, i.e., for any $u, v \in V_1$, we have $(u,v) \notin E$. We define two versions of the independent set problem.

1. optimization-version: given an undirected graph $G = (V,E)$, to seek an independent set $V_1$ of $G$ such that $V_1$ is maximized; we denote this problem as $IS(G)$;

2. decision-version: given an undirected graph $G = (V,E)$ and integer $k \geq 0$, to decide if there exists an independent set $V_1$ of $G$ such that $|V_1| \geq k$; we denote this problem as $IS(G,k)$.

There is a simple connection between independent set and vertex cover, stated below. Can you spot why? See Figure 2 for an example.




$V_1 = \{e, d, c\}$ is an independent set of $G$

$V \setminus V_1 = \{a, b, f\}$ is a vertex cover of $G$

Figure 2: Relationship between independent set and vertex cover.

**Claim 3.** Let $G = (V, E)$ be an undirected graph. We have $V_1 \subset V$ is an independent set of $G$ if and only if $V \setminus V_1$ is a vertex cover of $G$.

Above claim immediately suggests the following corollaries.

**Corollary 1.** $V_1^* \subset V$ is one maximum independent set of an undirected graph $G$ if and only if $V \setminus V_1^*$ is one minimum vertex cover of $G$.

**Corollary 2.** $G$ has an independent set with $k$ vertices if and only if $G$ has a vertex cover with $|V| - k$ vertices.

According to Corollary 2, we have $VC(G,k) \leq_p IS(G,k)$, as we can design an algorithm for $VC(G,k)$ with one statement "return solver-for-$IS(G, |V| - k)$". Note that, this actually proves that $IS(G,k)$ is NP-complete, given that $VC(G,k)$ is NP-complete and $IS(G,k)$ is in NP.

## NP-hard

**Definition 2** (NP-hard)**.** Let $X$ be a problem. We say $X$ is NP-hard, if there exists an NP-complete problem $Y$ such that $Y \leq_p X$.

Intutively, a problem is said to be NP-hard, if it is at least as hard as the hardest problem in NP (i.e., an NP-complete problem). Note that NP-hard problems are not restricted to decision problems. This makes it convinent to use. For examples, we can say the (optimization-version of) vertex cover problem is NP-hard, the (optimization-version of) independent set problem is NP-hard, the TSP problem is NP-hard, etc. Of course, all NP-complete problems are also NP-hard. In fact, by definitions, we have

**Fact 1.** NP $\cap$ NP-hard = NPC, where we also use "NP-hard" to represent the set of all NP-hard problems.

Above fact suggests the following two possibilities among P, NP, NPC, and NP-hard. See Figure 3.


Figure 3: Two possibilities among P, NP, NPC, and NP-hard.