Algorithm Design XIV

NP Problem II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

NP-Completeness

Hard Problems, Easy Problems

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, Horn SAT
Traveling salesman problem	Minimum spanning tree
Longest path	Shortest path
3D matching	Bipartite matching
Knapsack	Unary knapsack
Independent set	Independent set on trees
Integer linear programming	Linear programming
Rudrata path	Euler path
Balanced cut	Minimum cut

NP

Recall a search problem is defined by:

Recall a search problem is defined by:

1. An efficient checking algorithm C, taking as input the given `instance` I, a `solution` S, and outputs `true` iff S is a solution I.
2. The running time of $C(I, S)$ is bounded by a polynomial in $|I|$.

NP

Recall a search problem is defined by:

1. An efficient checking algorithm C, taking as input the given `instance` I, a `solution` S, and outputs `true` iff S is a solution I.
2. The running time of $C(I, S)$ is bounded by a polynomial in $|I|$.

We denote the class of all search problems by NP.

NP问题就是对于所有搜索问题的统称，所以证明一个问题是一个NP问题，最先的思路就应该是从定义出发，证明是一个搜索问题。

P

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

P

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

- I has a solution, the algorithm returns such a solution;
- I has no solution, the algorithm correctly reports so.

P

An algorithm that takes as input an instance I and has a running time polynomial in $|I|$.

- I has a solution, the algorithm returns such a solution;
- I has no solution, the algorithm correctly reports so.

The class of all search problems that can be solved in polynomial time is denoted P.

search problem需要有一个高效的搜索算法，但是不一定有一个多项式时间的求解方法。
P问题指的就是那些存在多项式时间解法的那些search problem。

课上的一个补充：SAT问题一般认为是一个搜索问题，那么如何将这个问题转化为一个优化问题？
SAT<-->判断是否存在一组指派使得一个合取范式为T<-->sesrch problem，很好验证
<-->Max SAT<-->Opt 问题<-->若已经知道一个合取范式不存在一个指派使得所有子句都为true，则**寻找一个指派，使得有最多的子句能够被满足**。

Why P and NP

P: polynomial time

NP: nondeterministic polynomial time

P $\neq$ NP

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

A formal proof of a mathematical assertion is written out in excruciating detail, it can be checked mechanically, by an efficient algorithm and is therefore in NP.

$P \neq NP$

P不等于NP，并且一般认为P是NP的一个子集。对于一个P问题，也一定是NP的，因为既然每一个P问题都存在一个多项式时间的算法来计算出一个结果/输出无解，那么在进行对应的search problem的check的时候，只需要运行这个算法，将算法结果与给定的结果比较，就知道成不成立了，所以也是一个NP问题。

证明问题<-->NP问题

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

A formal proof of a mathematical assertion is written out in excruciating detail, it can be checked mechanically, by an efficient algorithm and is therefore in NP.

So if $P = NP$, there would be an efficient method to prove any theorem, thus eliminating the need for mathematicians!

Solve One and All Solved

Even if we believe $\mathsf{P} \neq \mathsf{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

Solve One and All Solved

Even if we believe $\mathsf{P} \neq \mathsf{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

Solve One and All Solved

Even if we believe $P \neq NP$, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the hard problems in previous lecture exactly the same problem, the hardest search problems in NP.

Solve One and All Solved

Even if we believe $\mathrm{P} \neq \mathrm{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

(reduction:归约)

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the hard problems in previous lecture exactly the same problem, the hardest search problems in NP.

If one of them has a polynomial time algorithm, then every problem in NP has a polynomial time algorithm.

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

Reduction Between Search Problems

SHANGHAI JIAO TONG UNIVERSITY

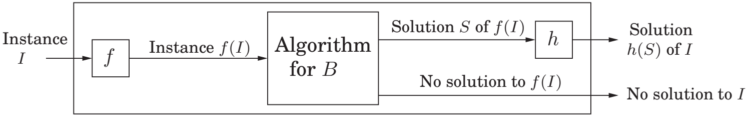
对于一个归约A->B，一般认为A问题**不会比B问题还要难**。

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A.



The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

对于归约A->B，有如下几种常见情况：
1.A未知->B未知，但是B有一个求解算法，即是一个**求解**过程。
2.A已知并且很难->B未知，由于A不会比B更难，所以B也是一个hard的problem，即是一个**证明**过程。
3.A未知->B已知，且已经知道B是一个hard的问题，那么可以知道A也可能很hard，这是一个**评估**过程。

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \rightarrow B$$

- If we can solve B efficiently, then we can also solve A efficiently.

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

If $A \to B$ and $B \to C$, then $A \to C$.

NP-Completeness

Definition

A NP problem is NP-complete if all other NP problems reduce to it.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

If even one NP-complete problem is in P, then P = NP.

Reductions to NP-Complete

SHANGHAI JIAO TONG UNIVERSITY

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

If even one NP-complete problem is in P, then P = NP.

If a problem A is NP-complete, a new NP problem B is proved to be NP-complete, by reducing A to B. （说明NPC问题已经是一种最难的问题了）

证明一个问题是一个NPC问题的思路：
1.先证明这个问题是一个NP问题，通过定义来证明是一个搜索问题
(注意这里最好详细说一下为何能在多项式时间内完成验证)；
2.之后将一个已知的NPC问题归约到这个问题上面。

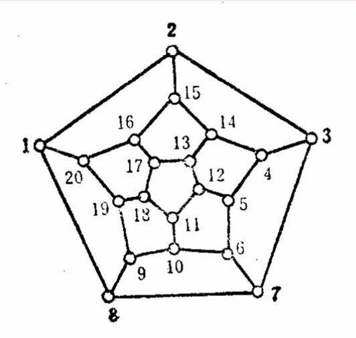
Reduction

RUDRATA PATH $\rightarrow$ RUDRATA CYCLE

Rudrata Cycle

Rudrata Cycle

Given a graph, find a cycle that visits each vertex exactly once.



RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE

A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE

A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

Q: Is it possible that RUDRATA CYCLE is easier than RUDRATA (s,t)-PATH?

RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE

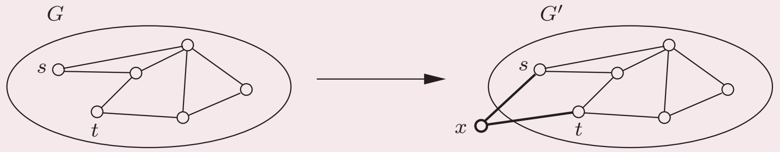
A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

Q: Is it possible that RUDRATA CYCLE is easier than RUDRATA (s,t)-PATH?

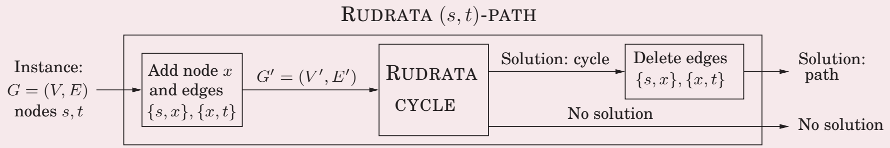
The reduction maps an `instance` G of RUDRATA (s,t)-PATH into an `instance` G' of RUDRATA CYCLE as follows: G' is G with an additional vertex x and two new edges $\{s,x\}$ and $\{x,t\}$.

注：这里不能直接将s和t连接起来。因为不能保证rudrata cycle一定会用到这条路径，如果没有用到这条路径，那么即使将边st移除之后仍然是一个**环**。
而添加一个**原图之外**的点x就不会有这个问题，因为为了遍历到点x一定会走sx和xt。

RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE



Rudrata (s,t)-path $\rightarrow$ Rudrata cycle



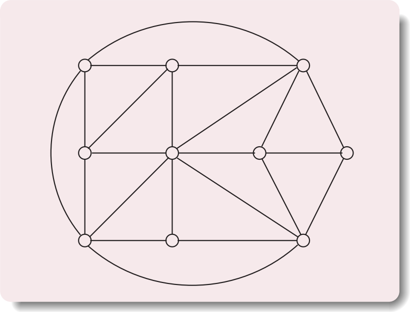
3SAT $\rightarrow$ Independent set

3 SAT

The instances of 3SAT, is set of clauses, each with three or fewer literals.

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

Independent Set



INDEPENDENT SET: Given a graph G and an integer g, find g vertices, no two of which have an edge between them.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

The choices must be consistent, if we choose $\overline{x}$ in one clause, we cannot choose x in another.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

The choices must be consistent, if we choose $\overline{x}$ in one clause, we cannot choose x in another.

Solution: put an edge between any two vertices that correspond to opposite literals.

独立集中的任意两点之间都没有边，所以为了不让相反的一组指派在一个独立集中，加一条边

Clause

Clause

Represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$.

Clause

Represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$.

Because a triangle has its three vertices maximally connected, and thus forces to pick only one of them for the independent set.

3SAT $\to$ INDEPENDENT SET

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

3SAT $\to$ INDEPENDENT SET

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.

3SAT $\to$ INDEPENDENT SET

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.
- Additional edges between any two vertices that represent opposite literals.

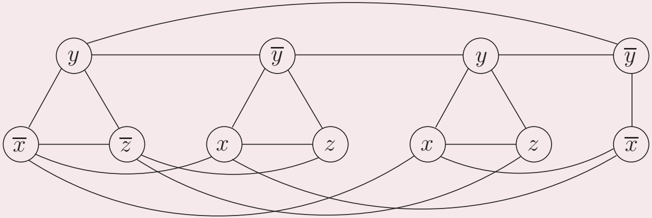
3SAT → Independent Set

Given an `instance` I of 3SAT, create an `instance` (G, g) of Independent Set as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.
- Additional edges between any two vertices that represent opposite literals.
- The goal g is set to the number of clauses.

这个过程就是归约算法中的instance I ofA-->instance I' of B的过程。
对于如何将B的实例结果转化为A的实例的结果也很容易，即如果B的实例有一个解，即存在一个点个数正好为g的一个独立集，那么原来的3SAT问题有解，反之则没解。

3SAT $\rightarrow$ Independent set



$$(\overline{x} \vee y \vee \overline{z})(x \vee \overline{y} \vee z)(x \vee y \vee z)(\overline{x} \vee \overline{y})$$

SAT $\rightarrow$ 3SAT

SAT $\rightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

SAT $\rightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, except that any clause with more than three literals,

$$(a_1 \vee a_2 \vee \ldots \vee a_k)$$

is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)(\overline{y_1} \vee a_3 \vee y_2)(\overline{y_2} \vee a_4 \vee y_3) \ldots (\overline{y_{k-3}} \vee a_{k-1} \vee a_k)$$

where the y_i's are new variables.

SAT $\rightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, except that any clause with more than three literals,

$$(a_1 \vee a_2 \vee \ldots \vee a_k)$$

is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)(\overline{y_1} \vee a_3 \vee y_2)(\overline{y_2} \vee a_4 \vee y_3)\ldots(\overline{y_{k-3}} \vee a_{k-1} \vee a_k)$$

where the y_i's are new variables.

The reduction is in polynomial and I' is equivalent to I in terms of satisfiability.

SAT $\to$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

SAT $\to$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \iff \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`. (<==)

<==的证明：假设右面式子成立的时候，左面的式子不成立。那么说明任意一个a均为false。那么由于右面的式子是成立的，那么y1一定true，y2一定是true，。。。。y(k-3)一定是true。但是这样的话就会导致最后一个子句为false，就与已知矛盾。所以右面式子成立的时候左面的式子一定成立。

SAT $\rightarrow$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \dots, a_k$ must be `true`. Otherwise y_1 would have to be `true`, which would in turn force y_2 to be `true`, and so on.

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`. Otherwise y_1 would have to be `true`, which would in turn force y_2 to be `true`, and so on.

Conversely, if $(a_1 \vee a_2 \vee \ldots \vee a_k)$ is satisfied, then some a_i must be `true`. Set $y_1, \ldots, y_{i-2}$ to `true` and the rest to `false`.

==>的证明：若左面的式子成立，那么至少有一个a为true，设为a_i。假设其余的a均为false。那么就要把a_i前面的所有出现的y均设置为true(考虑第一个子句)，所有a_i后面出现的y均设置为false(考虑最后一个子句)。这样就可以使得右面的式子成立。

SAT $\rightarrow$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

SAT $\rightarrow$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

SAT → 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

Finally, add the clauses

$$(\overline{x_1} \vee x_2)(\overline{x_2} \vee x_3) \ldots (\overline{x_k} \vee x_1)$$

SAT $\rightarrow$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

下面这个公式要想满足，一定要使得所有的x均为true或者所有的x均为false，
而这就等价于**一个单独的变量**，因为一个单独的变量也只能取到true/false。
所以可以用下面的一组子句来表示某一个单一的变量。使用这个替换之后可以保证
每一个变量出现不超过3次，同时每一个文字x_i出现次数不会超过两次。(正/负)

Finally, add the clauses

$$(\overline{x_1} \vee x_2)(\overline{x_2} \vee x_3)\ldots(\overline{x_k} \vee x_1)$$

In the new formula no variable appears more than three times (and in fact, no literal appears more than twice).

RUDRATA CYCLE $\rightarrow$ TSP

Rudrata cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

Rudrata cycle → TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

If G has a RUDRATA CYCLE, then the same cycle is also a tour within the budget of the TSP `instance`.

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

If G has a RUDRATA CYCLE, then the same cycle is also a tour within the budget of the TSP `instance`.

If G has no RUDRATA CYCLE, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$.

RUDRATA CYCLE $\rightarrow$ TSP

If $\alpha = 1$, then all distances are either 1 or 2, and so this `instance` of the TSP satisfies the triangle inequality: if i, j, k are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}$$

RUDRATA CYCLE $\rightarrow$ TSP

If $\alpha = 1$, then all distances are either 1 or 2, and so this `instance` of the TSP satisfies the triangle inequality: if i, j, k are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}$$

This is a special case of the TSP which is in a certain sense easier, since it can be efficiently approximated.

If α is large, then the resulting `instance` of the TSP may not satisfy the triangle inequality, and has another important property.

Rudrata cycle $\rightarrow$ TSP

If α is large, then the resulting `instance` of the TSP may not satisfy the triangle inequality, and has another important property.

This important gap property implies that, unless P = NP, no approximation algorithm is possible.

ANY PROBLEM $\rightarrow$ SAT

home reading!

Homework

Homework

Assignment 6(1 week). Exercises 8.3, 8.9, 8.14 and 8.19.