Kolkata Algorithms Short Course: III-IV Parallel/Streamable Algorithms and Equation Solving

Kenneth W. Regan
University at Buffalo (SUNY)

University of Calcutta, 3 August 2016

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.
- Sorting the list can be done in $O(n \log n)$ time—e.g. by Heapsort as described—then any duplicates will be adjacent.

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.
- Sorting the list can be done in $O(n \log n)$ time—e.g. by Heapsort as described—then any duplicates will be adjacent.
- So overall time is $O(n \log n)$. Recall that n times any power of $\log n$ gives *quasilinear time*.

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.
- Sorting the list can be done in $O(n \log n)$ time—e.g. by Heapsort as described—then any duplicates will be adjacent.
- So overall time is $O(n \log n)$. Recall that n times any power of $\log n$ gives *quasilinear time*.
- A second substantial efficiency of sorting is that its work can be distributed.

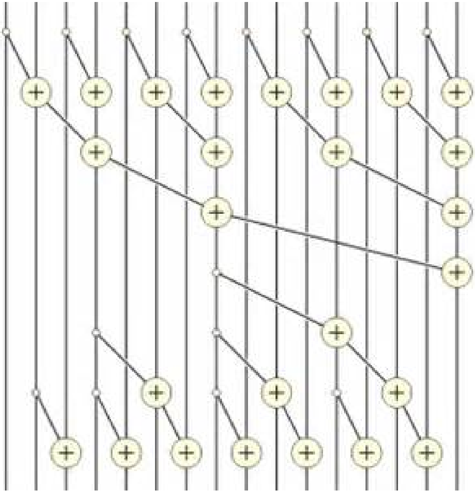
Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.
- Sorting the list can be done in $O(n \log n)$ time—e.g. by Heapsort as described—then any duplicates will be adjacent.
- So overall time is $O(n \log n)$. Recall that n times any power of $\log n$ gives *quasilinear time*.
- A second substantial efficiency of sorting is that its work can be distributed.
- One sense of this is that sorting is *streamable*, especially Mergesort.

Sorting and Sub-Quadratic Time

- Given a list of n words—figure the list is very long—how time does it take to determine whether there are two or more occurrences of the very same word?
- Comparing every pair of words would take time of order n^2.
- Sorting the list can be done in $O(n \log n)$ time—e.g. by Heapsort as described—then any duplicates will be adjacent.
- So overall time is $O(n \log n)$. Recall that n times any power of $\log n$ gives *quasilinear time*.
- A second substantial efficiency of sorting is that its work can be distributed.
- One sense of this is that sorting is *streamable*, especially Mergesort.
- Another is that sorting has Boolean circuits a power of $\log n$ in *depth*.

Parallel Prefix Sum (PPS): Depth $2 \log n$



Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.
- Applying an operation at every point in a list is called *map*.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.
- Applying an operation at every point in a list is called *map*.
- Thus $(a_1, \quad a_1 \odot a_2, \quad a_1 \odot a_2 \odot a_3, \quad \ldots, \quad a_1 \odot a_2 \odot \cdots \odot a_n)$ is the "Map-Reduce" of the list.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.
- Applying an operation at every point in a list is called *map*.
- Thus $(a_1, \quad a_1 \odot a_2, \quad a_1 \odot a_2 \odot a_3, \quad \ldots, \quad a_1 \odot a_2 \odot \cdots \odot a_n)$ is the "Map-Reduce" of the list.
- Wikipedia says this "inspired" the much more general "MapReduce" architecture for cloud computing, which retains the idea of a poly-log(n)-width stream.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.
- Applying an operation at every point in a list is called *map*.
- Thus $(a_1, \quad a_1 \odot a_2, \quad a_1 \odot a_2 \odot a_3, \quad \ldots, \quad a_1 \odot a_2 \odot \cdots \odot a_n)$ is the "Map-Reduce" of the list.
- Wikipedia says this "inspired" the much more general "MapReduce" architecture for cloud computing, which retains the idea of a poly-log(n)-width stream.What it must *avoid* is $\Omega(n)$-width random access.

Generalization

- Looking top-to-bottom, we have $O(\log n)$-delay parallel processing.
- Looking left-to-right, we we have an $O(\log n)$-width *stream*.
- The same algorithm works for any binary associative operation $\odot$.
- The act of computing a list $(a_1, a_2, \ldots, a_n)$ into a value $a_1 \odot a_2 \odot \cdots \odot a_n$ is called *reduce*.
- Applying an operation at every point in a list is called *map*.
- Thus $(a_1, \quad a_1 \odot a_2, \quad a_1 \odot a_2 \odot a_3, \quad \ldots, \quad a_1 \odot a_2 \odot \cdots \odot a_n)$ is the “Map-Reduce” of the list.
- Wikipedia says this “inspired” the much more general “MapReduce” architecture for cloud computing, which retains the idea of a poly-log(n)-width stream.What it must *avoid* is $\Omega(n)$-width random access. Sorting and PPS give a toolkit.

Finite State Machine Example

- A *finite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.

Finite State Machine Example

- A *fnite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.

Finite State Machine Example

- A *fnite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.

Finite State Machine Example

- A *finite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.
- At end when machine halts in a state q the machine appends $\phi(q)$ to its output; if q is not an accepting state then $\phi(q) =$ "Cancel!"

Finite State Machine Example

- A *finite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.
- At end when machine halts in a state q the machine appends $\phi(q)$ to its output; if q is not an accepting state then $\phi(q) =$ "Cancel!"
- Examples: "zoom in," "zoom out," parity check, running sums...

Finite State Machine Example

- A *finite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.
- At end when machine halts in a state q the machine appends $\phi(q)$ to its output; if q is not an accepting state then $\phi(q) =$ "Cancel!"
- Examples: "zoom in," "zoom out," parity check, running sums. . .
- Execution problem: given a string x, compute $T(x)$.

Finite State Machine Example

- A *finite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.
- At end when machine halts in a state q the machine appends $\phi(q)$ to its output; if q is not an accepting state then $\phi(q) =$ "Cancel!"
- Examples: "zoom in," "zoom out," parity check, running sums...
- Execution problem: given a string x, compute $T(x)$.
- Streaming is easy, but parallel execution is harder: how do we know ahead of time what state T will be in towad the end?

Finite State Machine Example

- A *fnite state transducer* (FST) is a Turing machine $T = (Q, \Sigma, \delta, \rho, s, \phi)$ with a read-only input tape and a write-only *output tape*.
- Besides $\delta : Q \times \Sigma \to Q$ we have the *output function* $\rho : Q \times \Sigma \to \Sigma^*$ and a *final-output function* $\phi : Q \to \Sigma^*$.
- Output can be more than one char or can be empty; it is fixed into the code of T.
- At end when machine halts in a state q the machine appends $\phi(q)$ to its output; if q is not an accepting state then $\phi(q) =$ "Cancel!"
- Examples: "zoom in," "zoom out," parity check, running sums...
- Execution problem: given a string x, compute $T(x)$.
- Streaming is easy, but parallel execution is harder: how do we know ahead of time what state T will be in towad the end?
- Answer: use PPS to compose the maps $g_c(q) = \delta(q, c)$ for each character; $g_c \odot g_d =$ take q to $g_d(g_c(q))$ [show on board].

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row.

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?
- We will do $O(\log n)$ recursive passes over the lists.

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?
- We will do $O(\log n)$ recursive passes over the lists.
- Key idea is that if you reverse B into B', then the list A, B' is *bitonic*—like a valley.

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?
- We will do $O(\log n)$ recursive passes over the lists.
- Key idea is that if you reverse B into B', then the list A, B' is *bitonic*—like a valley.
- Strangely, compare first half of A with first half of B' not B, then second halves.

Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?
- We will do $O(\log n)$ recursive passes over the lists.
- Key idea is that if you reverse B into B', then the list A, B' is *bitonic*—like a valley.
- Strangely, compare first half of A with first half of B' not B, then second halves.
- The four outputs of size $n/2$ are bitonic so we can recurse.

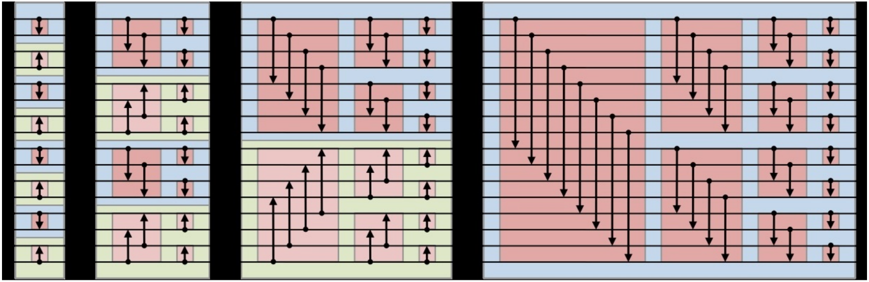
Batcher's Bitonic Merge and Sort

- Given two already-sorted lists $A = a_1 \leq a_2 \leq \cdots \leq a_n$ and $B = b_1 \leq b_2 \leq \cdots \leq b_n$ of equal length n, you want to merge them into one sorted list.
- A *comparator gate* g maps $g(a, b) = (b, a)$ if $b < a$, else (a, b).
- Stream is easy if you can "pause" the flow of one of the lists—in case the other list has many lesser items in a row. But what if not, and what about parallel?
- We will do $O(\log n)$ recursive passes over the lists.
- Key idea is that if you reverse B into B', then the list A, B' is *bitonic*—like a valley.
- Strangely, compare first half of A with first half of B' not B, then second halves.
- The four outputs of size $n/2$ are bitonic so we can recurse.
- Gives Mergesort in $O(n \log n)$ time with $O((\log n)^2)$ depth.

Python code from Wikipedia

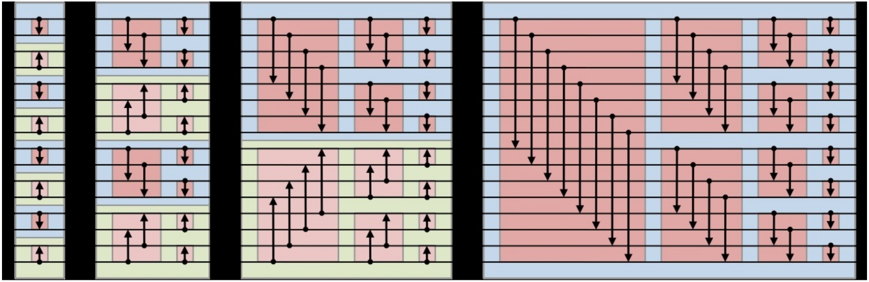
```
def bitonic_merge(up, x): # assume input x is bitonic
    if len(x) == 1:
        return x
    else:
        bitonic_compare(up, x)
        first = bitonic_merge(up, x[:len(x) / 2])
        second = bitonic_merge(up, x[len(x) / 2:])
        return first + second

def bitonic_compare(up, x):
    dist = len(x) / 2
    for i in range(dist):
        if (x[i] > x[i+dist]) == up:
            x[i], x[i+dist] = x[i+dist], x[i] #swap
```

Picture (from Wikipedia)



Theorem: Every decision problem or function in nondeterministc logspace can be processed in parallel by circuits of $n^{O(1)}$ size and $O((\log n)^2)$ depth.

Picture (from Wikipedia)



Theorem: Every decision problem or function in nondeterministc logspace can be processed in parallel by circuits of $n^{O(1)}$ size and $O((\log n)^2)$ depth.

Thus one reason to care about the theoretical distinction of the "BFS class" is being able to make better parallel/cloud-friendly algorithms.

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.
- The w clause prevents just taking $x = u$ or $x = v$ so the answers ae different.

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.
- The w clause prevents just taking $x = u$ or $x = v$ so the answers ae different.
- But it goes away from integers. . .

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.
- The w clause prevents just taking $x = u$ or $x = v$ so the answers ae different.
- But it goes away from integers...
- *General question:* When are equations solvable?

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.
- The w clause prevents just taking $x = u$ or $x = v$ so the answers ae different.
- But it goes away from integers...
- *General question:* When are equations solvable? in reals or integers?

Solving Arithmetical Equations

A famous example:

$$\begin{aligned} z &= x^3 + y^3; \\ z &= u^3 + v^3; \\ w * (x - u) * (x - v) &= 1. \end{aligned}$$

- About 100 years ago, the English mathematician G.H. Hardy hailed a taxicab with Srinivasa Ramanujan that had the number $z = 1,729$.
- Ramanujan solved it instantly with $x = 1$, $y = 12$, $u = 9$, $v = 10$.
- The w clause prevents just taking $x = u$ or $x = v$ so the answers ae different.
- But it goes away from integers...
- *General question:* When are equations solvable? in reals or integers? or in 0-1 values only?

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete.**
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u$ NAND v.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u$ NAND v.
- Equation form: $w = 1 - uv$.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u \text{ NAND } v$.
- Equation form: $w = 1 - uv$.
- If the NAND gate has multiple outgoing wires w_i, add equations $w_i = w$.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u$ NAND v.
- Equation form: $w = 1 - uv$.
- If the NAND gate has multiple outgoing wires w_i, add equations $w_i = w$.
- General 3-clause $(u \vee \bar{v} \vee w)$ becomes equation $(1 - u)v(1 - w) = 0$.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u$ NAND v.
- Equation form: $w = 1 - uv$.
- If the NAND gate has multiple outgoing wires w_i, add equations $w_i = w$.
- General 3-clause $(u \vee \bar{v} \vee w)$ becomes equation $(1 - u)v(1 - w) = 0$.
- Add equations $u^2 - u = 0$, $v^2 - v = 0$, and $w^2 - w = 0$ to limit to 0-1 solutions.

A Big Obstacle!—?

- Let's recall the logical *Satisfiability* problem from Day 2, only this time for 3CNF formulas not 2CNF.
- We showed *2SAT* is easy to solve—indeed in the BFS class. But *3SAT* is **NP-complete**.
- Typical 3CNF formula: $(u \vee w) \wedge (v \vee w) \wedge (\bar{u} \vee \bar{v} \vee \bar{w})$.
- Expresses the correct behavior of a NAND gate: $w = u \text{ NAND } v$.
- Equation form: $w = 1 - uv$.
- If the NAND gate has multiple outgoing wires w_i, add equations $w_i = w$.
- General 3-clause $(u \vee \bar{v} \vee w)$ becomes equation $(1-u)v(1-w) = 0$.
- Add equations $u^2 - u = 0$, $v^2 - v = 0$, and $w^2 - w = 0$ to limit to 0-1 solutions.
- Thus equation solving is **NP-hard**.

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$.

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$. What does this *mean*?

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$. What does this *mean*?
- It means you have a yes/no problem where if the answer is yes, an inspired guess will give an answer that you can easily prove.

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$. What does this *mean*?
- It means you have a yes/no problem where if the answer is yes, an inspired guess will give an answer that you can easily prove.
- If the answer is no, there may be no short proof—that's OK.

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$. What does this *mean*?
- It means you have a yes/no problem where if the answer is yes, an inspired guess will give an answer that you can easily prove.
- If the answer is no, there may be no short proof—that's OK.
- For 3SAT the inspired quess is an assignment $a \in \{0, 1\}^n$ making $\phi(a) = \texttt{true}$.

NP-Hard and Complete

- Recall we defined $\mathsf{NP} = \mathsf{NTIME}[n^{O(1)}]$. What does this *mean*?
- It means you have a yes/no problem where if the answer is yes, an inspired guess will give an answer that you can easily prove.
- If the answer is no, there may be no short proof—that's OK.
- For 3SAT the inspired quess is an assignment $a \in \{0,1\}^n$ making $\phi(a) = \mathtt{true}$.
- For equations the inspired guess is a solution; it is easy to check unless the math is too $\mathbb{C}$omplex.

NP-Hard and Complete

- Recall we defined NP $=$ NTIME$[n^{O(1)}]$. What does this *mean*?
- It means you have a yes/no problem where if the answer is yes, an inspired guess will give an answer that you can easily prove.
- If the answer is no, there may be no short proof—that's OK.
- For 3SAT the inspired quess is an assignment $a \in \{0,1\}^n$ making $\phi(a) = \mathtt{true}$.
- For equations the inspired guess is a solution; it is easy to check unless the math is too $\mathbb{C}$omplex.
- So 3SAT is in NP and basically so is equation solving—over $\{0,1\}$-solutions anyway.

Definition. A decision problem B is *NP-hard* if for all problems A in NP there is a polynomial-time computable translation function f such that for all inputs x of problem A, the string $y = f(x)$ is an equivalent input of problem B. And B is *NP-complete* if also B is in NP.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in$ NP there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \mathrm{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \ldots, x_n$ of x and $y_1, \ldots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \mathrm{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \ldots, x_n$ of x and $y_1, \ldots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.
- The function $f(x)$ will produce a 3CNF formula ϕ such that $x \in A$ (meaning the answer for x is 'yes') if and only if ϕ is satisfiable.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \mathrm{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \dots, x_n$ of x and $y_1, \dots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.
- The function $f(x)$ will produce a 3CNF formula ϕ such that $x \in A$ (meaning the answer for x is 'yes') if and only if ϕ is satisfiable.
- Most of ϕ doesn't involve x—only at the end we will substitute the actual bits of x for the variables $x_1, \dots, x_n$.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \mathrm{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \dots, x_n$ of x and $y_1, \dots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.
- The function $f(x)$ will produce a 3CNF formula ϕ such that $x \in A$ (meaning the answer for x is 'yes') if and only if ϕ is satisfiable.
- Most of ϕ doesn't involve x—only at the end we will substitute the actual bits of x for the variables $x_1, \dots, x_n$.
- The left-over variables in ϕ will be $y_1, \dots, y_m$ and extra *wire variables* $u, v, w, \dots$ including a variable w_o for the output value.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \mathrm{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \ldots, x_n$ of x and $y_1, \ldots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.
- The function $f(x)$ will produce a 3CNF formula ϕ such that $x \in A$ (meaning the answer for x is 'yes') if and only if ϕ is satisfiable.
- Most of ϕ doesn't involve x—only at the end we will substitute the actual bits of x for the variables $x_1, \ldots, x_n$.
- The left-over variables in ϕ will be $y_1, \ldots, y_m$ and extra *wire variables* $u, v, w, \ldots$ including a variable w_o for the output value.
- Each of these variables can appear negated: $\bar{y}_1, \ldots, \bar{y}_m, \bar{u}, \bar{v}, \bar{w}$ etc.

Cook-Levin Theorem: 3SAT is NP-Complete

- Given $A \in \text{NP}$ there is a *deterministic* TM M that verifies the relation "y is a lucky guess for $x \in A$" in polynomial time.
- The memory map for M includes the bits $x_1, \ldots, x_n$ of x and $y_1, \ldots, y_m$ of potential verifying strings y, where $m = n^{O(1)}$.
- The function $f(x)$ will produce a 3CNF formula ϕ such that $x \in A$ (meaning the answer for x is 'yes') if and only if ϕ is satisfiable.
- Most of ϕ doesn't involve x—only at the end we will substitute the actual bits of x for the variables $x_1, \ldots, x_n$.
- The left-over variables in ϕ will be $y_1, \ldots, y_m$ and extra *wire variables* $u, v, w, \ldots$ including a variable w_o for the output value.
- Each of these variables can appear negated: $\bar{y}_1, \ldots, \bar{y}_m, \bar{u}, \bar{v}, \bar{w}$ etc.
- The key is what we covered in day 2: the memory map of M can be converted into Boolean circuits C_n, one for each n (and the corresponding m) such that M accepts (x, y) if and only if $C_n(x, y) = 1$. We can build C_n using only NAND gates.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.
- And add to ϕ the "singleton clause" (w_o) for the output wire—to satisfy ϕ, this must have value 1.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.
- And add to ϕ the "singleton clause" (w_o) for the output wire—to satisfy ϕ, this must have value 1.
- Finally substitute the bits of x for $x_1, \ldots, x_n$. This finishes $\phi = f(x)$.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.
- And add to ϕ the "singleton clause" (w_o) for the output wire—to satisfy ϕ, this must have value 1.
- Finally substitute the bits of x for $x_1, \ldots, x_n$. This finishes $\phi = f(x)$.
- Then ϕ is satisfiable $\iff$ there is a setting of $y_1, \ldots, y_m$ and all other u_g, v_g, w_k variables that satisfies $\phi \iff$ there is a y that M verifies for $x \iff x \in A$.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \dots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.
- And add to ϕ the "singleton clause" (w_o) for the output wire—to satisfy ϕ, this must have value 1.
- Finally substitute the bits of x for $x_1, \dots, x_n$. This finishes $\phi = f(x)$.
- Then ϕ is satisfiable $\iff$ there is a setting of $y_1, \dots, y_m$ and all other u_g, v_g, w_k variables that satisfies $\phi \iff$ there is a y that M verifies for $x \iff x \in A$.
- Since the memory map has size at worst quadratic in the time and space by M, which are both $n^{O(1)}$, and since the rules for building ϕ are so regular, $f(x) = \phi$ is computed in polynomial time.

Finishing the Proof

- For each NAND gate g, let u_g and v_g be its two incoming wires (these can be inputs x_i or y_j) and $w_1, \ldots, w_\ell$ its output wires.
- Add to ϕ the clauses $(u_g \vee w_k) \wedge (v_g \vee w_k) \wedge (\bar{u}_g \vee \bar{v}_g \vee \bar{w}_g)$ for each k, $1 \leq k \leq \ell$.
- And add to ϕ the "singleton clause" (w_o) for the output wire—to satisfy ϕ, this must have value 1.
- Finally substitute the bits of x for $x_1, \ldots, x_n$. This finishes $\phi = f(x)$.
- Then ϕ is satisfiable $\iff$ there is a setting of $y_1, \ldots, y_m$ and all other u_g, v_g, w_k variables that satisfies ϕ $\iff$ there is a y that M verifies for x $\iff$ $x \in A$.
- Since the memory map has size at worst quadratic in the time and space by M, which are both $n^{O(1)}$, and since the rules for building ϕ are so regular, $f(x) = \phi$ is computed in polynomial time.
- So 3SAT is NP-hard, and since it is in NP, it is NP-complete. $\square$

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation

$$1 - u_g v_g - w_g = 0.$$

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation
$$1 - u_g v_g - w_g = 0.$$
- For any other outgoing wires w_k, use $w_g - w_k = 0$ to set them all equal.

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation
$$1 - u_g v_g - w_g = 0.$$
- For any other outgoing wires w_k, use $w_g - w_k = 0$ to set them all equal.
- And we have $1 - w_o = 0$ for the output wire and the "Boolean equations" $u_g^2 - u_g = 0$ (etc.) for every variable. That's it.

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation
$$1 - u_g v_g - w_g = 0.$$
- For any other outgoing wires w_k, use $w_g - w_k = 0$ to set them all equal.
- And we have $1 - w_o = 0$ for the output wire and the "Boolean equations" $u_g^2 - u_g = 0$ (etc.) for every variable. That's it.
- This makes the sokving problem for simple equations likewise NP-complete. □

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation
$$1 - u_g v_g - w_g = 0.$$
- For any other outgoing wires w_k, use $w_g - w_k = 0$ to set them all equal.
- And we have $1 - w_o = 0$ for the output wire and the "Boolean equations" $u_g^2 - u_g = 0$ (etc.) for every variable. That's it.
- This makes the sokving problem for simple equations likewise NP-complete. □

The equations in this proof are indeed *very* simple—degree 2 for the $u_g v_g$ terms and the Boolean equations.

And for Equation Solving...

- To finish that equation solving is NP-hard: for each NAND gate g with incoming wires u_g, v_g and outgoing wire w_g we give the equation
$$1 - u_g v_g - w_g = 0.$$
- For any other outgoing wires w_k, use $w_g - w_k = 0$ to set them all equal.
- And we have $1 - w_o = 0$ for the output wire and the "Boolean equations" $u_g^2 - u_g = 0$ (etc.) for every variable. That's it.
- This makes the sokving problem for simple equations likewise NP-complete. □

The equations in this proof are indeed *very* simple—degree 2 for the $u_g v_g$ terms and the Boolean equations. Does this really mean that solving them is hard in practice?

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.
- Newer tide: reduce problems *to* SAT and *to* equation solving because many individual instances terminate acceptably quickly.

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.
- Newer tide: reduce problems *to* SAT and *to* equation solving because many individual instances terminate acceptably quickly.
- General reason: the formulas/equations used in the hardness proof are specialized enough that many real-world instances avoid their "region of hardness."

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.
- Newer tide: reduce problems *to* SAT and *to* equation solving because many individual instances terminate acceptably quickly.
- General reason: the formulas/equations used in the hardness proof are specialized enough that many real-world instances avoid their "region of hardness."
- Indeed, *randomly* generated instances of 3SAT with n variables and m clauses tend to be easily solved.

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.
- Newer tide: reduce problems *to* SAT and *to* equation solving because many individual instances terminate acceptably quickly.
- General reason: the formulas/equations used in the hardness proof are specialized enough that many real-world instances avoid their "region of hardness."
- Indeed, *randomly* generated instances of 3SAT with n variables and m clauses tend to be easily solved. If m is larger than a certain window the formula tends to have an easily-seen contradiction.

A Practical Sea-Change

- Classic course and attitude: reduce *from* (3)SAT to other problems to show they are *hard*.
- Newer tide: reduce problems *to* SAT and *to* equation solving because many individual instances terminate acceptably quickly.
- General reason: the formulas/equations used in the hardness proof are specialized enough that many real-world instances avoid their "region of hardness."
- Indeed, *randomly* generated instances of 3SAT with n variables and m clauses tend to be easily solved. If m is larger than a certain window the formula tends to have an easily-seen contradiction. if m is smaller than the window, then "standard greedy" tends to work.

A Standard Greedy Heuristic Algorithm

```
set<Clause> TODO = clauses(phi);
set<Variable> FREE = {x_1,...,x_n}
while (TODO and FREE are both nonempty) {
   Choose the x_i or -x_i in most clauses TODO;
   Set a_i = true or false accordingly;
   TODO \= {newly satisfied clauses};
   FREE \= {x_i};
}
if (empty TODO) {
   return satisfying assignment (a_1,...,a_n);
} else {
   fail; maybe re-try with randomised x_i choices?
}
```

A Standard Greedy Heuristic Algorithm

```
set<Clause> TODO = clauses(phi);
set<Variable> FREE = {x_1,...,x_n}
while (TODO and FREE are both nonempty) {
   Choose the x_i or -x_i in most clauses TODO;
   Set a_i = true or false accordingly;
   TODO \= {newly satisfied clauses};
   FREE \= {x_i};
}
if (empty TODO) {
   return satisfying assignment (a_1,...,a_n);
} else {
   fail; maybe re-try with randomised x_i choices?
}
```

Current “SAT Solvers” use more-sophisticated heuristics.

Equation Solvers Use a Hammer

Represent a given set of pure-arithmetic equations abstractly as

$$\begin{aligned} p_1(z_1, \dots, z_n) &= 0; \\ p_2(z_1, \dots, z_n) &= 0; \\ \vdots &= 0; \\ p_s(z_1, \dots, z_n) &= 0; \end{aligned}$$

where each p_i is a multi-variable polynomial. Now observe:

Equation Solvers Use a Hammer

Represent a given set of pure-arithmetic equations abstractly as

$$\begin{aligned} p_1(z_1,\dots,z_n) &= 0;\\ p_2(z_1,\dots,z_n) &= 0;\\ \vdots &= 0;\\ p_s(z_1,\dots,z_n) &= 0; \end{aligned}$$

where each p_i is a multi-variable polynomial. Now observe:

For any polynomials $q_1,\dots,q_s$ in the same variables $\vec{z}$, the polynomial

$$r(\vec{z}) = q_1(\vec{z})p_1(\vec{z}) + q_2(\vec{z})p_2(\vec{z}) + \cdots q_s(\vec{z})p_s(\vec{z})$$

must also be equated to 0. Call it an “algebraic consequence.”

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.
- Ditto the lack of a solution: David Hilbert proved in his *Nullstellensatz* ("Theorem About Zeroes") that if the equations have *no* solution *over the complex numbers*, then the constant 1 (which would give the contradictory equation $1 = 0$) is an algebraic consequence!

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.
- Ditto the lack of a solution: David Hilbert proved in his *Nullstellensatz* ("Theorem About Zeroes") that if the equations have *no* solution *over the complex numbers*, then the constant 1 (which would give the contradictory equation $1 = 0$) is an algebraic consequence!
- *Buchberger's Algorithm* (BA) compiles a certain exhaustive list of non-redundant consequence called a *Gr'́obner basis.*

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal*.
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.
- Ditto the lack of a solution: David Hilbert proved in his *Nullstellensatz* ("Theorem About Zeroes") that if the equations have *no* solution *over the complex numbers*, then the constant 1 (which would give the contradictory equation $1 = 0$) is an algebraic consequence!
- *Buchberger's Algorithm* (BA) compiles a certain exhaustive list of non-redundant consequence called a *Gr'́obner basis*.
- Often the basis finds simplified equations that allow solutions to be read off.

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.
- Ditto the lack of a solution: David Hilbert proved in his *Nullstellensatz* ("Theorem About Zeroes") that if the equations have *no* solution *over the complex numbers*, then the constant 1 (which would give the contradictory equation $1 = 0$) is an algebraic consequence!
- *Buchberger's Algorithm* (BA) compiles a certain exhaustive list of non-redundant consequence called a *Gr´obner basis.*
- Often the basis finds simplified equations that allow solutions to be read off.
- Sometimes BA runs for time $\approx 2^{d^n}$ where d is the max degre of the given polynomials $p_1, \ldots, p_s$, which in worst case is double-exponentially horrible.

Idea of Buchberger's Algorithm

- Technically the algebraic consequences form a *polynomial ideal.*
- Some $r(\vec{z}$ have *cancellations* that make solutions easier to see.
- Ditto the lack of a solution: David Hilbert proved in his *Nullstellensatz* ("Theorem About Zeroes") that if the equations have *no* solution *over the complex numbers*, then the constant 1 (which would give the contradictory equation $1 = 0$) is an algebraic consequence!
- *Buchberger's Algorithm* (BA) compiles a certain exhaustive list of non-redundant consequence called a *Gr'́obner basis.*
- Often the basis finds simplified equations that allow solutions to be read off.
- Sometimes BA runs for time $\approx 2^{d^n}$ where d is the max degree of the given polynomials $p_1, \ldots, p_s$, which in worst case is double-exponentially horrible.
- But in many cases it finishes quickly enough, so people use it. . .

Example: Graph 3-Coloring to SAT and EQNs

Example: Graph 3-Coloring to SAT and EQNs

[show SAT on board, with “atoms” and then without.]

Example: Graph 3-Coloring to SAT and EQNs

[show SAT on board, with "atoms" and then without.]

[show equations on board, maybe run them?]

Example: Graph 3-Coloring to SAT and EQNs

[show SAT on board, with “atoms” and then without.]

[show equations on board, maybe run them?]

[show Buchberger’s notes]