I. Sorting Networks

Thomas Sauerwald
Introduction to Sorting Networks

Batcher’s Sorting Network

Counting Networks

Load Balancing on Graphs
Overview: Sorting Networks

(Serial) Sorting Algorithms

- we already know several (comparison-based) sorting algorithms: Insertion sort, Bubble sort, Merge sort, Quick sort, Heap sort
- execute one operation at a time
- can handle arbitrarily large inputs
- sequence of comparisons is not set in advance

Sorting Networks

- only perform comparisons
- can only handle inputs of a fixed size
- sequence of comparisons is set in advance
- Comparisons can be performed in parallel

Simple concept, but surprisingly deep and complex theory!

Allows to sort $n$ numbers in sublinear time!
A sorting network is a comparison network which works correctly (that is, it sorts every input).

A comparison network consists solely of wires and comparators:
- comparator is a device with, on given two inputs, $x$ and $y$, returns two outputs $x'$ and $y'$
- wire connect output of one comparator to the input of another
- special wires: $n$ input wires $a_1, a_2, \ldots, a_n$ and $n$ output wires $b_1, b_2, \ldots, b_n$

Convention: use the same name for both a wire and its value.

Comparison Network

- **Comparator**
  - $x \xrightarrow{\text{min}} x'$
  - $y \xrightarrow{\text{max}} y'$

Figure 27.1  (a) A comparator with inputs $x$ and $y$ and outputs $x'$ and $y'$. (b) The same comparator, drawn as a single vertical line. Inputs $x = 7$, $y = 3$ and outputs $x' = 3$, $y' = 7$ are shown.
Example of a Comparison Network (Figure 27.2)

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network!

Depth of a wire:
- Input wire has depth 0.
- If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$.

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2)

Interconnections between comparators must be acyclic

A horizontal line represents a sequence of distinct wires. This network is in fact a sorting network!

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths \( d_x \) and \( d_y \), then outputs have depth \( \max\{d_x, d_y\} + 1 \)

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic.

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2)

Interconnections between comparators must be **acyclic** ✓

A horizontal line represents a sequence of distinct wires.

This network is in fact a sorting network!

Depth of a wire:
- Input wire has depth 0
- If a comparator has two inputs of depths \(d_x\) and \(d_y\), then outputs have depth \(\max\{d_x, d_y\} + 1\)

Maximum depth of an output wire equals total running time.

Interconnections between comparators must be acyclic ✓
Example of a Comparison Network (Figure 27.2)

Interconnections between comparators must be *acyclic*

Tracing back a path must never cycle back on itself and go through the same comparator twice.
Example of a Comparison Network (Figure 27.2)

This network is in fact a sorting network!
Example of a Comparison Network (Figure 27.2)

- **Depth of a wire:**
  - Input wire has depth 0
  - If a comparator has two inputs of depths $d_x$ and $d_y$, then outputs have depth $\max\{d_x, d_y\} + 1$

The network is in fact a sorting network!
Zero-One Principle

Zero-One Principle: A sorting networks works correctly on arbitrary inputs if it works correctly on binary inputs.

Lemma 27.1

If a comparison network transforms the input \( a = \langle a_1, a_2, \ldots, a_n \rangle \) into the output \( b = \langle b_1, b_2, \ldots, b_n \rangle \), then for any monotonically increasing function \( f \), the network transforms \( f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \) into \( f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle \).

\[
\begin{align*}
f(x) & \quad \text{min}(f(x), f(y)) = f(\text{min}(x, y)) \\
f(y) & \quad \text{max}(f(x), f(y)) = f(\text{max}(x, y))
\end{align*}
\]

Figure 27.4 The operation of the comparator in the proof of Lemma 27.1. The function \( f \) is monotonically increasing.
Zero-One Principle

Zero-One Principle: A sorting networks works correctly on arbitrary inputs if it works correctly on binary inputs.

Lemma 27.1

If a comparison network transforms the input \( a = \langle a_1, a_2, \ldots, a_n \rangle \) into the output \( b = \langle b_1, b_2, \ldots, b_n \rangle \), then for any monotonically increasing function \( f \), the network transforms \( f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle \) into \( f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle \).

Theorem 27.2 (Zero-One Principle)

If a comparison network with \( n \) inputs sorts all \( 2^n \) possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.
Proof of the Zero-One Principle

Theorem 27.2 (Zero-One Principle)

If a comparison network with $n$ inputs sorts all $2^n$ possible sequences of 0’s and 1’s correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof:

- For the sake of contradiction, suppose the network does not correctly sort.
- Let $a = \langle a_1, a_2, \ldots, a_n \rangle$ be the input with $a_i < a_j$, but the network places $a_j$ before $a_i$ in the output.
- Define a monotonically increasing function $f$ as:

$$f(x) = \begin{cases} 0 & \text{if } x \leq a_i, \\ 1 & \text{if } x > a_i. \end{cases}$$

- Since the network places $a_j$ before $a_i$, by the previous lemma $\Rightarrow f(a_j)$ is placed before $f(a_i)$.
- But $f(a_j) = 1$ and $f(a_i) = 0$, which contradicts the assumption that the network sorts all sequences of 0’s and 1’s correctly.
Some Basic (Recursive) Sorting Networks

$n$-wire Sorting Network

These are Sorting Networks, but with depth $\Theta(n)$. 
Outline

Introduction to Sorting Networks

Batcher’s Sorting Network

Counting Networks

Load Balancing on Graphs
Bitonic Sequences

A sequence is **bitonic** if it monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing.

Sequences of one or two numbers are defined to be bitonic.

**Examples:**

- \(\langle 1, 4, 6, 8, 3, 2 \rangle\) ✓
- \(\langle 6, 9, 4, 2, 3, 5 \rangle\) ✓
- \(\langle 9, 8, 3, 2, 4, 6 \rangle\) ✓
- \(\langle 4, 5, 7, 1, 2, 6 \rangle\)
- **binary sequences:** \(0^i1^j0^k\), or, \(1^i0^j1^k\), for \(i, j, k \geq 0\).
Towards Bitonic Sorting Networks

**Half-Cleaner**

A half-cleaner is a comparison network of depth 1 in which input wire $i$ is compared with wire $i + n/2$ for $i = 1, 2, \ldots, n/2$.

We always assume that $n$ is even.

**Lemma 27.3**

If the input to a half-cleaner is a bitonic sequence of 0’s and 1’s, then the output satisfies the following properties:

- both the top half and the bottom half are bitonic,
- every element in the top is not larger than any element in the bottom,
- at least one half is clean.

Proof

The comparison network HALF-CLEANER compares input wires $i$ and $i + n/2$ for $i = 1, 2, \ldots, n/2$. Without loss of generality, suppose that the input is of the form 00...011...100...0. (The situation in which the input is of the form 11...100...011...1 is symmetric.) There are possible cases depending upon the block of consecutive 0’s or 1’s in which the midpoint falls, and one of these cases (the one in which the midpoint occurs in the block of 1’s) is further split into two cases. The four cases are shown in Figure 27.8. In each case shown, the lemma holds.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i 1^j 0^k$, for some $i, j, k \geq 0$. 

![Diagram of sorting network](image)

(a) Cases in which the division occurs in the middle subsequence of 1's.

(b) Cases in which the division occurs in a subsequence of 0's. For all cases, every element in the top half of the output is at least as small as every element in the bottom half, both halves are bitonic, and at least one half is clean.

This suggests a recursive approach, since it now suffices to sort the top and bottom half separately.
Proof of Lemma 27.3

W.l.o.g. assume that the input is of the form $0^i1^j0^k$, for some $i, j, k \geq 0$.

This suggests a recursive approach, since it now suffices to sort the top and bottom half separately.
The Bitonic Sorter

Figure 27.9  The comparison network BITONIC-SORTER\([n]\), shown here for \(n = 8\). (a) The recursive construction: HALF-CLEANER\([n]\) followed by two copies of BITONIC-SORTER\([n/2]\) that operate in parallel. (b) The network after unrolling the recursion. Each half-cleaning is shaded. Sample zero-one values are shown on the wires.

Recursive Formula for depth \(D(n)\):

\[
D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + 1 & \text{if } n = 2^k.
\end{cases}
\]

Henceforth we will always assume that \(n\) is a power of 2.

BITONIC-SORTER\([n]\) has depth \(\log n\) and sorts any zero-one bitonic sequence.

Exercises

27.3-1 How many zero-one bitonic sequences of length \(n\) are there?
Merging Networks

- can merge two sorted input sequences into one sorted output sequences
- will be based on a modification of BITONIC-SORTER[n]

Basic Idea:
- consider two given sequences \( X = 0000111, \ Y = 00001111 \)
- concatenating \( X \) with \( Y^R \) (the reversal of \( Y \)) \( \Rightarrow 0000111111110000 \)

This sequence is bitonic!

Hence in order to merge the sequences \( X \) and \( Y \), it suffices to perform a bitonic sort on \( X \) concatenated with \( Y^R \).
Construction of a Merging Network (1/2)

- Given two sorted sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \)
- We know it suffices to bitonically sort \( \langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1} \rangle \)
- Recall: first half-cleaner of BITONIC-SORTER\( [n] \) compares \( i \) and \( n/2 + i \)

\[ \Rightarrow \] First part of MERGER\( [n] \) compares inputs \( i \) and \( n - i \) for \( i = 1, 2, \ldots, n/2 \)
- Remaining part is identical to BITONIC-SORTER\( [n] \)

![Diagram](image)

Lemma 27.3 still applies, since the reversal of a bitonic sequence is bitonic.

**Figure 27.10** Comparing the first stage of MERGER\( [n] \) with HALF-CLEANER\( [n] \), for \( n = 8 \).
(a) The first stage of MERGER\( [n] \) transforms the two monotonic input sequences \( \langle a_1, a_2, \ldots, a_{n/2} \rangle \) and \( \langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n \rangle \) into two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n \rangle \).
(b) The equivalent operation for HALF-CLEANER\( [n] \). The bitonic input sequence \( \langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1} \rangle \) is transformed into the two bitonic sequences \( \langle b_1, b_2, \ldots, b_{n/2} \rangle \) and \( \langle b_n, b_{n-1}, \ldots, b_{n/2+1} \rangle \).
Construction of a Merging Network (2/2)

Figure 27.11  A network that merges two sorted input sequences into one sorted output sequence. The network MERGER[n] can be viewed as BITONIC-SORTER[n] with the first half-cleaner altered to compare inputs $i$ and $n - i + 1$ for $i = 1, 2, \ldots, n/2$. Here, $n = 8$. (a) The network decomposed into the first stage followed by two parallel copies of BITONIC-SORTER[n/2]. (b) The same network with the recursion unrolled. Sample zero-one values are shown on the wires, and the stages are shaded.
Construction of a Sorting Network

Main Components

1. **BITONIC-SORTER**\([n]\)
   - sorts any bitonic sequence
   - depth \(\log n\)

2. **MERGER**\([n]\)
   - merges two sorted input sequences
   - depth \(\log n\)

Batcher’s Sorting Network

- **SORTER**\([n]\) is defined recursively:
  - If \(n = 2^k\), use two copies of **SORTER**\([n/2]\) to sort two subsequences of length \(n/2\) each. Then merge them using **MERGER**\([n]\).
  - If \(n = 1\), network consists of a single wire.

*can be seen as a parallel version of merge sort*
Unrolling the Recursion (Figure 27.12)

Recursion for $D(n)$:

$$D(n) = \begin{cases} 
0 & \text{if } n = 1, \\
D(n/2) + \log n & \text{if } n = 2^k.
\end{cases}$$

Solution: $D(n) = \Theta(\log^2 n)$.

**SORTER**[$n$] has depth $\Theta(\log^2 n)$ and sorts any input.
A Glimpse at the AKS Network

There exists a sorting network with depth $O(\log n)$.

Quite elaborate construction, and involves huge constants.

**Perfect Halver**

A perfect halver is a comparator network that, given any input, places the $n/2$ smaller keys in $b_1, \ldots, b_{n/2}$ and the $n/2$ larger keys in $b_{n/2+1}, \ldots, b_n$.

Perfect halver of depth $\log_2 n$ exist $\Rightarrow$ yields sorting networks of depth $\Theta((\log n)^2)$.

**Approximate Halver**

An $(n, \epsilon)$-approximate halver, $\epsilon < 1$, is a comparator network that for every $k = 1, 2, \ldots, n/2$ places at most $\epsilon k$ of its $k$ smallest keys in $b_{n/2+1}, \ldots, b_n$ and at most $\epsilon k$ of its $k$ largest keys in $b_1, \ldots, b_{n/2}$.

We will prove that such networks can be constructed in constant depth!
Expander Graphs

A bipartite \((n, d, \mu)\)-expander is a graph with:
- \(G\) has \(n\) vertices (\(n/2\) on each side)
- the edge-set is the union of \(d\) matchings
- For every subset \(S \subseteq V\) being in one part,
  \[|N(S)| \geq \min\{\mu \cdot |S|, n/2 - |S|\}\]

Expander Graphs:
- **probabilistic construction** “easy”: take \(d\) (disjoint) random matchings
- **explicit construction** is a deep mathematical problem with ties to number theory, group theory, combinatorics etc.
- **many applications** in networking, complexity theory and coding theory
From Expanders to Approximate Halvers

I. Sorting Networks

Batcher's Sorting Network
From Expanders to Approximate Halvers

I. Sorting Networks

Batcher's Sorting Network
From Expanders to Approximate Halvers

I. Sorting Networks

Batcher’s Sorting Network
From Expanders to Approximate Halvers

I. Sorting Networks

Batcher's Sorting Network
Existence of Approximate Halvers

Proof:

- \( X \) := wires with the \( k \) smallest inputs
- \( Y \) := wires in lower half with \( k \) smallest outputs
- For every \( u \in \mathcal{N}(Y) \): \( \exists \) comparator \((u, v)\)
- Let \( u_t, v_t \) be their keys after the comparator
  Let \( u_d, v_d \) be their keys at the output
- Note that \( v_d \in Y \subseteq X \)
- Further: \( u_d \leq u_t \leq v_t \leq v_d \Rightarrow u_d \in X \)
- Since \( u \) was arbitrary:
  \[
  |Y| + |\mathcal{N}(Y)| \leq k.
  \]
- Since \( G \) is a bipartite \((n, d, \mu)\)-expander:
  \[
  |Y| + |\mathcal{N}(Y)| \geq |Y| + \min\{\mu |Y|, n/2 - |Y|\}
  = \min\{(1 + \mu)|Y|, n/2\}.
  \]
- Combining the two bounds above yields:
  \[
  (1 + \mu)|Y| \leq k.
  \]
- The same argument shows that at most \( \epsilon \cdot k \), 
  \( \epsilon := 1/(\mu + 1) \), of the \( k \) largest input keys are
  placed in \( b_1, \ldots, b_{n/2} \). 

I. Sorting Networks
Batcher’s Sorting Network
Donald E. Knuth (Stanford)

“Batcher’s method is much better, unless $n$ exceeds the total memory capacity of all computers on earth!”

Richard J. Lipton (Georgia Tech)

“The AKS sorting network is galactic: it needs that $n$ be larger than $2^{78}$ or so to finally be smaller than Batcher’s network for $n$ items.”
Siblings of Sorting Network

- **Sorting Networks**
  - sorts any input of size $n$
  - special case of **Comparison Networks**

- **Switching (Shuffling) Networks**
  - creates a random permutation of $n$ items
  - special case of **Permutation Networks**

- **Counting Networks**
  - balances any stream of tokens over $n$ wires
  - special case of **Balancing Networks**
Outline

Introduction to Sorting Networks

Batcher’s Sorting Network

Counting Networks

Load Balancing on Graphs
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Values could represent addresses in memories or destinations on an interconnection network.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)
Counting Network

Distributed Counting

Processors collectively assign successive values from a given range.

Values could represent addresses in memories or destinations on an interconnection network.

Balancing Networks

- constructed in a similar manner like sorting networks
- instead of comparators, consists of balancers
- balancers are asynchronous flip-flops that forward tokens from its inputs to one of its two outputs alternately (top, bottom, top, ...)

Number of tokens differs by at most one
1. Let $x_1, x_2, \ldots, x_n$ be the number of tokens (ever received) on the designated input wires.

2. Let $y_1, y_2, \ldots, y_n$ be the number of tokens (ever received) on the designated output wires.

3. In a quiescent state: $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$.

4. A counting network is a balancing network with the step-property:

   $$0 \leq y_i - y_j \leq 1 \text{ for any } i < j.$$ 

---

**Bitonic Counting Network**: Take Batcher’s Sorting Network and replace each comparator by a balancer.
Correctness of the Bitonic Counting Network

Facts

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have \[ \sum_{i=1}^{n/2} x_{2i-1} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor, \text{ and } \sum_{i=1}^{n/2} x_{2i} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil \]
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Key Lemma

Consider a MERGER$[n]$. Then if the inputs $x_1, \ldots, x_{n/2}$ and $x_{n/2+1}, \ldots, x_n$ have the step property, then so does the output $y_1, \ldots, y_n$.

Proof (by induction on $n$)

- Case $n = 2$ is clear, since MERGER$[2]$ is a single balancer
Correctness of the Bitonic Counting Network

Facts

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have \( \sum_{i=1}^{n/2} x_{2i-1} = \left\lceil \frac{1}{2} \sum_{i=1}^{n} x_i \right\rceil \) and \( \sum_{i=1}^{n/2} x_{2i} = \left\lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \right\rfloor \).
2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).
3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

Proof (by induction on \( n \))

- Case \( n = 2 \) is clear, since MERGER[2] is a single balancer.
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the MERGER\([n/2]\) subnetworks.
Correctness of the Bitonic Counting Network

Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) have the step property. Then:

1. We have \( \sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil \) and \( \sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor \)

2. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \), then \( x_i = y_i \) for \( i = 1, \ldots, n \).

3. If \( \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1 \), then \( \exists! j = 1, 2, \ldots, n \) with \( x_j = y_j + 1 \) and \( x_i = y_i \) for \( j \neq i \).

I. Sorting Networks
   Counting Networks

---

Image: Diagram of a bitonic counting network with inputs \( x_1 \) to \( x_8 \) and outputs \( z_1 \) to \( z_4 \).

Proof (by induction on \( n \))

- Case \( n = 2 \) is clear, since MERGER[2] is a single balancer.
- \( n > 2 \): Let \( z_1, \ldots, z_{n/2} \) and \( z'_1, \ldots, z'_{n/2} \) be the outputs of the MERGER[n/2] subnetworks.
Correctness of the Bitonic Counting Network

Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ have the step property. Then:

1. We have $\sum_{i=1}^{n/2} x_{2i-1} = \lceil \frac{1}{2} \sum_{i=1}^{n} x_i \rceil$, and $\sum_{i=1}^{n/2} x_{2i} = \lfloor \frac{1}{2} \sum_{i=1}^{n} x_i \rfloor$
2. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$, then $x_i = y_i$ for $i = 1, \ldots, n$.
3. If $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i + 1$, then $\exists ! j = 1, 2, \ldots, n$ with $x_j = y_j + 1$ and $x_i = y_i$ for $j \neq i$.

Proof (by induction on $n$)

- Case $n = 2$ is clear, since MERGER[2] is a single balancer
- $n > 2$: Let $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ be the outputs of the MERGER[$n/2$] subnetworks
- IH $\Rightarrow$ $z_1, \ldots, z_{n/2}$ and $z'_1, \ldots, z'_{n/2}$ have the step property
- Let $Z := \sum_{i=1}^{n/2} z_i$ and $Z' := \sum_{i=1}^{n/2} z'_i$
- F1 $\Rightarrow$ $Z = \lceil \frac{1}{2} \sum_{i=1}^{n/2} x_i \rceil + \lfloor \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \rfloor$ and $Z' = \lfloor \frac{1}{2} \sum_{i=1}^{n/2} x_i \rfloor + \lceil \frac{1}{2} \sum_{i=n/2+1}^{n} x_i \rceil$
- Case 1: If $Z = Z'$, then F2 implies the output of MERGER[$n$] is $y_i = z_1 + \lfloor (i-1)/2 \rfloor$ ✓
- Case 2: If $|Z - Z'| = 1$, F3 implies $z_i = z'_i$ for $i = 1, \ldots, n/2$ except a unique $j$ with $z_j \neq z'_j$.

Balancer between $z_j$ and $z'_j$ will ensure that the step property holds.
Counting can be done as follows:

Add local counter to each output wire $i$, to assign consecutive numbers $i, i+n, i+2\cdot n, ...$,
Counting can be done as follows: Add local counter to each output wire $i$, to assign consecutive numbers $i, i + n, i + 2 \cdot n, \ldots$
A Periodic Counting Network [Aspnes, Herlihy, Shavit, JACM 1994]

Consists of $\log n$ BLOC[K][n] networks each of which has depth $\log n$
If a network is a counting network, then it is also a sorting network.

Proof.

- Let $C$ be a counting network, and $S$ be the corresponding sorting network.
- Consider an input sequence $a_1, a_2, \ldots, a_n \in \{0, 1\}^n$ to $S$.
- Define an input $x_1, x_2, \ldots, x_n \in \{0, 1\}^n$ to $C$ by $x_i = 1$ iff $a_i = 0$.
- $C$ is a counting network $\Rightarrow$ all ones will be routed to the lower wires.
- $S$ corresponds to $C$ $\Rightarrow$ all zeros will be routed to the lower wires.
- By the Zero-One Principle, $S$ is a sorting network.

![Diagram showing C and S networks]
Outline

Introduction to Sorting Networks

Batcher’s Sorting Network

Counting Networks

Load Balancing on Graphs
Communication Models: Diffusion vs. Matching

\[ M = \begin{pmatrix} \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} & 0 \\ \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & \frac{1}{3} \end{pmatrix} \]

\[ M(t) = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \]
Smoothness of the Load Distribution

- let $x \in \mathbb{R}^n$ be a load vector
- $x^t \in \mathbb{R}^n$ be a load vector at round $t$
- $\bar{x}$ denotes the average load

**Metrics**

- $\ell_2$-norm: $\Phi^t = \sqrt{\sum_{i=1}^{n}(x_i^t - \bar{x})^2}$
- makespan: $\max_{i=1}^{n} x_i^t$
- discrepancy: $\max_{i=1}^{n} x_i^t - \min_{i=1}^{n} x_i$.

For this example:

- $\Phi^t = \sqrt{0^2 + 0^2 + 3.5^2 + 0.5^2 + 1^2 + 1^2 + 1.5^2 + 0.5^2} = \sqrt{17}$
- $\max_{i=1}^{n} x_i^t = 6.5$
- $\max_{i=1}^{n} x_i^t - \min_{i=1}^{n} x_i^t = 5$
Given an undirected, connected graph $G = (V, E)$ and a diffusion parameter $\alpha > 0$, the diffusion matrix $M$ is defined as follows:

$$M_{ij} = \begin{cases} 
\alpha & \text{if } (i, j) \in E, \\
1 - \alpha \deg(i) & \text{if } i = j, \\
0 & \text{otherwise}.
\end{cases}$$

Further let $\gamma(M) := \max_{\mu_i \neq 1} |\mu_i|$, where $\mu_1 = 1 > \mu_2 \geq \cdots \geq \mu_n \geq -1$ are the eigenvalues of $M$.

How to choose $\alpha$ for a $d$-regular graph?

- $\alpha = \frac{1}{d}$ may lead to oscillation (if graph is bipartite)
- $\alpha = \frac{1}{d+1}$ ensures convergence
- $\alpha = \frac{1}{2d}$ ensures convergence (and all eigenvalues of $M$ are non-negative)

This can be also seen as a random walk on $G$!

**First-Order Diffusion:** Load vector $x^t$ satisfies

$$x^t = M \cdot x^{t-1}.$$
\( \gamma(M) \approx 1 - \frac{1}{n^2} \) for 1D grid

\( \gamma(M) \approx 1 - \frac{1}{n} \) for 2D grid

\( \gamma(M) \approx 1 - \frac{1}{n^{2/3}} \) for 3D grid

\( \gamma(M) \approx 1 - \frac{1}{\log n} \) for Hypercube

\( \gamma(M) < 1 \) for Random Graph

\( \gamma(M) \approx 0 \) for Complete Graph

\( \gamma(M) \in (0, 1] \) measures connectivity of \( G \)
Diffusion on a Ring

After iteration 2:

After iteration 3:

After iteration 4:

After iteration 5:

After iteration 20:
Diffusion on a Ring

after iteration 1:
Diffusion on a Ring

after iteration 20:

![Diagram of a ring network with labeled nodes after iteration 20]
Convergence of the Quadratic Error (Upper Bound)

Let $\gamma(M) := \max_{\mu_i \neq 1} |\mu_i|$, where $\mu_1 = 1 > \mu_2 \geq \cdots \geq \mu_n \geq -1$ are the eigenvalues of $M$. Then for any iteration $t$,

$$\Phi^t \leq \gamma(M)^2t \cdot \Phi^0.$$

**Proof:**

- Let $e^t = x^t - \bar{x}$, where $\bar{x}$ is the column vector with all entries set to $\bar{x}$
- Express $e^t$ through the orthogonal basis given by the eigenvectors of $M$:
  $$e^t = \alpha_1 \cdot v_1 + \alpha_2 \cdot v_2 + \cdots + \alpha_n \cdot v_n = \sum_{i=2}^{n} \alpha_i \cdot v_i.$$

- For the diffusion scheme,
  $$e^{t+1} = Me^t = M \cdot \left( \sum_{i=2}^{n} \alpha_i v_i \right) = \sum_{i=2}^{n} \alpha_i \mu_i v_i.$$
  $e^t$ is orthogonal to $v_1$

- Taking norms and using that the $v_i$’s are orthogonal,
  $$\|e^{t+1}\|_2 = \|Me^t\|_2 = \sum_{i=2}^{n} \alpha_i^2 \mu_i^2 \|v_i\|_2 \leq \gamma^2 \sum_{i=2}^{n} \alpha_i^2 \|v_i\|_2 = \gamma^2 \cdot \|e^t\|_2.$$
Convergence of the Quadratic Error (Lower Bound)

Lemma

For any eigenvalue $\mu_i$, $1 \leq i \leq n$, there is an initial load vector $x^0$ so that

$$\Phi_t = \mu_i^{2t} \cdot \Phi^0.$$

Proof:

- Let $x^0 = \bar{x} + v_i$, where $v_i$ is the eigenvector corresponding to $\mu_i$
- Then

$$e^t = Me^{t-1} = M^t e^0 = M^t v_i = \mu_i^t v_i,$$

and

$$\Phi^t = \|e^t\|_2 = \mu_i^{2t} \|v_i\|_2 = \mu_i^{2t} \Phi^0.$$
Outlook: Idealised versus Discrete Case

**Idealised Case**

\[
x^t = M \cdot x^{t-1}
= M^t \cdot x^0
\]

**Linear System**
- corresponds to Markov chain
- well-understood

**Non-Linear System**
- rounding of a Markov chain
- harder to analyze

**Discrete Case**

\[
y^t = M \cdot y^{t-1} + \Delta^t
= M^t \cdot y^0 + \sum_{s=1}^{t} M^{t-s} \cdot \Delta^s
\]

**Rounding Error**

Here load consists of integers that cannot be divided further.

Given any load vector \(x^0\), the number of iterations until \(x^t\) satisfies \(\Phi^t \leq \epsilon\) is at most \(\log(\Phi^0/\epsilon) / (1 - \gamma(M))\).

How close can it be made to the idealised case?
II. Matrix Multiplication

Thomas Sauerwald
Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Matrix Multiplication

Remember: If \( A = (a_{ij}) \) and \( B = (b_{ij}) \) are square \( n \times n \) matrices, then the matrix product \( C = A \cdot B \) is defined by

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.
\]

This definition suggests that \( n \cdot n^2 = n^3 \) arithmetic operations are necessary.

**SQUARE-MATRIX-MULTIPLY** \((A, B)\)

1. \( n = A\.rows \)
2. let \( C \) be a new \( n \times n \) matrix
3. **for** \( i = 1 \) **to** \( n \)
4. **for** \( j = 1 \) **to** \( n \)
5. \( c_{ij} = 0 \)
6. **for** \( k = 1 \) **to** \( n \)
7. \( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)
8. return \( C \)

**SQUARE-MATRIX-MULTIPLY** \((A, B)\) takes time \( \Theta(n^3) \).
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Assumption: \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A, B, \) and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]

Hence the equation \( C = A \cdot B \) becomes:

\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

This corresponds to the four equations:

\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{align*}

Each equation specifies two multiplications of \( n/2 \times n/2 \) matrices and the addition of their products.
Let $T(n)$ be the runtime of this procedure. Then:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1. 
\end{cases}$$

Solution: $T(n) = \Theta(8^{\log_2 n}) = \Theta(n^3)$

No improvement over the naive algorithm!
Divide & Conquer: Second Approach

**Idea:** Make the recursion tree less bushy by performing only 7 recursive multiplications of $n/2 \times n/2$ matrices.

**Strassen's Algorithm (1969)**

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$
4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.

Time for steps 1,2,4: $\Theta(n^2)$, hence $T(n) = 7 \cdot T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_7 7})$. 
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \]
\[ P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]
\[ P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + A_{22}B_{22} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}}
- A_{11}B_{22} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - \cancel{A_{22}B_{22}}
= A_{11}B_{11} + A_{12}B_{21}
\]
Conjecture: Does a quadratic-time algorithm exist?

Asymptotic Complexities:
- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
- $O(n^{2.796})$, Pan (1978)
- $O(n^{2.522})$, Schönhage (1981)
- $O(n^{2.517})$, Romani (1982)
- $O(n^{2.496})$, Coppersmith and Winograd (1982)
- $O(n^{2.479})$, Strassen (1986)
- $O(n^{2.376})$, Coppersmith and Winograd (1989)
- $O(n^{2.374})$, Stothers (2010)
- $O(n^{2.3728642})$, V. Williams (2011)
- $O(n^{2.3728639})$, Le Gall (2014)
- ...
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Memory Models

**Distributed Memory**
- Each processor has its private memory
- Access to memory of another processor via messages

**Shared Memory**
- Central location of memory
- Each processor has direct access

II. Matrix Multiplication Reminder: Multithreading
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use **concurrency platform** which coordinates all resources

Scheduling jobs, communication protocols, load balancing etc.

**Functionalities:**
- **spawn**
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread
- **sync**
  - wait until all spawned threads are done
- **parallel**
  - (optional) prefix to the standard loop **for**
  - each iteration is called in its own thread

Only logical parallelism, but not actual! Need a scheduler to map threads to processors.
Computing Fibonacci Numbers Recursively (Fig. 27.1)

Figure 27.1: The tree of recursive procedure instances when computing $F_6$. Each instance of $F_n$ with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

```plaintext
FIB(n)
1: if n<=1 return n
2: else x=FIB(n-1)
3: y=FIB(n-2)
4: return x+y
```

Very inefficient – exponential time!
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

- Without `spawn` and `sync` same pseudocode as before
- `spawn` does not imply parallel execution (depends on scheduler)

```
0: P-FIB(n)
1:   if n<=1 return n
2:   else x=spawn P-FIB(n-1)
3:     y=P-FIB(n-2)
4:     sync
5:     return x+y
```
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

\[
P\text{-FIB}(n)
\]

\begin{align*}
0: & \text{P-FIB}(n) \\
1: & \text{if } n \leq 1 \text{ return } n \\
2: & \text{else } x = \text{spawn P-FIB}(n-1) \\
3: & \quad y = \text{P-FIB}(n-2) \\
4: & \quad \text{sync} \\
5: & \quad \text{return } x + y
\end{align*}

Computation Dag \( G = (V, E) \)
- \( V \) set of threads (instructions/strands \textit{without parallel control})
- \( E \) set of dependencies

\[
\text{II. Matrix Multiplication Reminder: Multithreading 14}
\]
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \hspace{1em} if \( n \leq 1 \) return \( n \)
2: \hspace{1em} else \( x = \text{spawn} \ \texttt{P-FIB}(n-1) \)
3: \hspace{1em} \hspace{1em} \( y = \texttt{P-FIB}(n-2) \)
4: \hspace{1em} \hspace{1em} \hspace{1em} sync
5: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} return \( x + y \)

Total work \( \approx 17 \) nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Performance Measures

- **Work**
  - Total time to execute everything on single processor.

- **Graph Representation**
  - The graph shows a network of nodes and edges, with the sum of the nodes equal to 30.

- **Mathematical Notation**
  - \[ \sum = 30 \]
Performance Measures

**Work**
Total time to execute everything on single processor.

**Span**
Longest time to execute the threads along any path.

\[ \sum = 18 \]
Performance Measures

Work
Total time to execute everything on single processor.

Span
Longest time to execute the threads along any path.

If each thread takes unit time, span is the length of the critical path.

#nodes = 5
**Work Law and Span Law**

- $T_1 = \text{work}, \ T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

Running time actually also depends on scheduler etc.!

Work Law:

$$T_P \geq \frac{T_1}{P}$$

Time on $P$ processors can’t be shorter than if all work all time

$T_1 = 8, \ P = 2$

II. Matrix Multiplication Reminder: Multithreading
Work Law and Span Law

- $T_1 =$ work, $T_\infty =$ span
- $P =$ number of (identical) processors
- $T_P =$ running time on $P$ processors

Running time actually also depends on scheduler etc.!

**Work Law**

\[ T_P \geq \frac{T_1}{P} \]

Time on $P$ processors can’t be shorter than if all work all time

**Span Law**

\[ T_P \geq T_\infty \]

Time on $P$ processors can’t be shorter than time on $\infty$ processors

$T_\infty = 5$
Work Law and Span Law

- $T_1 = \text{work}, \quad T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

Running time actually also depends on scheduler etc.!

**Work Law**

$$T_P \geq \frac{T_1}{P}$$

Time on $P$ processors can’t be shorter than if all work all time

**Span Law**

$$T_P \geq T_\infty$$

Time on $P$ processors can’t be shorter than time on $\infty$ processors

- **Speed-Up**: $\frac{T_1}{T_P}$
  - Maximum Speed-Up bounded by $P$!
- **Parallelism**: $\frac{T_1}{T_\infty}$
  - Maximum Speed-Up for $\infty$ processors!
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Warmup: Matrix Vector Multiplication

Remember: Multiplying an \( n \times n \) matrix \( A = (a_{ij}) \) and \( n \)-vector \( x = (x_j) \) yields an \( n \)-vector \( y = (y_i) \) given by

\[
y_i = \sum_{j=1}^{n} a_{ij}x_j \quad \text{for } i = 1, 2, \ldots, n.
\]

**MAT-VEC(A, x)**

1. \( n = A.rows \)
2. let \( y \) be a new vector of length \( n \)
3. **parallel for** \( i = 1 \) to \( n \)
   4. \( y_i = 0 \)
5. **parallel for** \( i = 1 \) to \( n \)
   6. **for** \( j = 1 \) to \( n \)
   7. \( y_i = y_i + a_{ij}x_j \)
8. return \( y \)

The **parallel for**-loops can be used since different entries of \( y \) can be computed concurrently.

How can a compiler implement the **parallel for**-loop?
Implementing parallel for based on Divide-and-Conquer

\[
\text{MAT-Vec-Main-Loop}(A, x, y, n, i, i')
\]

1. \textbf{if} \( i = i' \)
2. \textbf{for} \( j = 1 \) to \( n \)
3. \hspace{1em} \( y_i = y_i + a_{ij}x_j \)
4. \textbf{else} \( \text{mid} = \left\lceil \frac{(i + i')}{2} \right\rceil \)
5. \hspace{1em} \textbf{spawn} \text{MAT-Vec-Main-Loop}(A, x, y, n, i, \text{mid})
6. \hspace{1em} \text{MAT-Vec-Main-Loop}(A, x, y, n, \text{mid} + 1, i')
7. \text{sync}

\[
T_1(n) = \Theta(n^2)
\]

\[
T_\infty(n) = \Theta(\log n) + \max_{1 \leq i \leq n} \text{iter}(n)
\]

\[= \Theta(n).\]

Work is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.

Span is the depth of recursive callings plus the maximum span of any of the \( n \) iterations.
Naive Algorithm in Parallel

P-SQUARE-MATRIX-MULTIPLY \((A, B)\)
1 \(n = A.rows\)
2 let \(C\) be a new \(n \times n\) matrix
3 \(\text{parallel for } i = 1 \text{ to } n\)
4 \(\text{parallel for } j = 1 \text{ to } n\)
5 \(c_{ij} = 0\)
6 \(\text{for } k = 1 \text{ to } n\)
7 \(c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}\)
8 \(\text{return } C\)

\(\text{P-SQUARE-MATRIX-MULTIPLY}(A, B)\) has work \(T_1(n) = \Theta(n^3)\) and span \(T_\infty(n) = \Theta(n)\).

The first two nested for-loops parallelise perfectly.
The Simple Divide & Conquer Approach in Parallel

\[ \text{P-Matrix-Multiply-Recursive}(C, A, B) \]

1. \( n = A.\text{rows} \)
2. \( \text{if } n == 1 \)
3. \( c_{11} = a_{11}b_{11} \)
4. \( \text{else let } T \text{ be a new } n \times n \text{ matrix} \)
5. \( \text{partition } A, B, C, \text{ and } T \text{ into } n/2 \times n/2 \text{ submatrices} \)
6. \( A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22}; \)
   \( \text{and } T_{11}, T_{12}, T_{21}, T_{22}; \) respectively
7. \( \text{spawn P-Matrix-Multiply-Recursive}(C_{11}, A_{11}, B_{11}) \)
8. \( \text{spawn P-Matrix-Multiply-Recursive}(C_{12}, A_{11}, B_{12}) \)
9. \( \text{spawn P-Matrix-Multiply-Recursive}(C_{21}, A_{21}, B_{11}) \)
10. \( \text{spawn P-Matrix-Multiply-Recursive}(C_{22}, A_{21}, B_{12}) \)
11. \( \text{spawn P-Matrix-Multiply-Recursive}(T_{11}, A_{12}, B_{21}) \)
12. \( \text{spawn P-Matrix-Multiply-Recursive}(T_{12}, A_{12}, B_{22}) \)
13. \( \text{spawn P-Matrix-Multiply-Recursive}(T_{21}, A_{22}, B_{21}) \)
14. \( \text{spawn P-Matrix-Multiply-Recursive}(T_{22}, A_{22}, B_{22}) \)
15. \( \text{sync} \)
16. \( \text{parallel for } i = 1 \text{ to } n \)
17. \( \text{parallel for } j = 1 \text{ to } n \)
   \[ c_{ij} = c_{ij} + t_{ij} \]

\[ \text{P-Matrix-Multiply-Recursive} \text{ has work } T_1(n) = \Theta(n^3) \text{ and span } T_\infty(n) = \Theta(\log^2 n). \]

\[ T_\infty(n) = T_\infty(n/2) + \Theta(\log n) \]
Strassen’s Algorithm (parallelised)

1. Partition each of the matrices into four \( n/2 \times n/2 \) submatrices

   This step takes \( \Theta(1) \) work and span by index calculations.

2. Create 10 matrices \( S_1, S_2, \ldots, S_{10} \). Each is \( n/2 \times n/2 \) and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with \( \Theta(n^2) \) work and \( \Theta(\log n) \) span using doubly nested parallel for loops.

3. Recursively compute 7 matrix products \( P_1, P_2, \ldots, P_7 \), each \( n/2 \times n/2 \)

   Recursively spawn the computation of the seven products.

4. Compute \( n/2 \times n/2 \) submatrices of \( C \) by adding and subtracting various combinations of the \( P_i \).

   Using doubly nested parallel for this takes \( \Theta(n^2) \) work and \( \Theta(\log n) \) span.

\[
T_1(n) = \Theta(n^{\log 7})
\]
\[
T_\infty(n) = \Theta(\log^2 n)
\]
Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof:

- Define a $3n \times 3n$ matrix $D$ by:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} \quad \Rightarrow \quad D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}.$$  

- Matrix $D$ can be constructed in $\Theta(n^2) = O(I(n))$ time,
- and we can invert $D$ in $O(I(3n)) = O(I(n))$ time.

$\Rightarrow$ We can compute $AB$ in $O(I(n))$ time.
The Other Direction

**Theorem 28.1 (Multiplication is no harder than Inversion)**

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Allows us to use Strassen’s Algorithm to invert a matrix!

**Theorem 28.2 (Inversion is no harder than Multiplication)**

Suppose we can multiply two $n \times n$ real matrices in time $M(n)$ and $M(n)$ satisfies the two regularity conditions $M(n + k) = O(M(n))$ for any $0 \leq k \leq n$ and $M(n/2) \leq c \cdot M(n)$ for some constant $c < 1/2$. Then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$.

Proof of this direction much harder (CLRS) – relies on properties of SPD matrices.
III. Linear Programming

Thomas Sauerwald
Outline

Introduction

Standard and Slack Forms

Formulating Problems as Linear Programs

Simplex Algorithm

Finding an Initial Solution
Introduction

Linear Programming (informal definition)

- maximize or minimize an objective, given limited resources and competing constraint
- constraints are specified as (in)equalities

Example: Political Advertising

- Imagine you are a politician trying to win an election
- Your district has three different types of areas: Urban, suburban and rural, each with, respectively, 100,000, 200,000 and 50,000 registered voters
- **Aim:** at least half of the registered voters in each of the three regions should vote for you
- **Possible Actions:** Advertise on one of the primary issues which are (i) building more roads, (ii) gun control, (iii) farm subsidies and (iv) a gasoline tax dedicated to improve public transit.
The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

Possible Solution:
- $20,000 on advertising to building roads
- $0 on advertising to gun control
- $4,000 on advertising to farm subsidies
- $9,000 on advertising to a gasoline tax

Total cost: $33,000
Towards a Linear Program

<table>
<thead>
<tr>
<th>policy</th>
<th>urban</th>
<th>suburban</th>
<th>rural</th>
</tr>
</thead>
<tbody>
<tr>
<td>build roads</td>
<td>−2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>gun control</td>
<td>8</td>
<td>2</td>
<td>−5</td>
</tr>
<tr>
<td>farm subsidies</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>gasoline tax</td>
<td>10</td>
<td>0</td>
<td>−2</td>
</tr>
</tbody>
</table>

The effects of policies on voters. Each entry describes the number of thousands of voters who could be won (lost) over by spending $1,000 on advertising support of a policy on a particular issue.

- $x_1 =$ number of thousands of dollars spent on advertising on building roads
- $x_2 =$ number of thousands of dollars spent on advertising on gun control
- $x_3 =$ number of thousands of dollars spent on advertising on farm subsidies
- $x_4 =$ number of thousands of dollars spent on advertising on gasoline tax

Constraints:

1. $−2x_1 + 8x_2 + 0x_3 + 10x_4 ≥ 50$
2. $5x_1 + 2x_2 + 0x_3 + 0x_4 ≥ 100$
3. $3x_1 − 5x_2 + 10x_3 − 2x_4 ≥ 25$

**Objective:** Minimize $x_1 + x_2 + x_3 + x_4$
The Linear Program

Linear Program for the Advertising Problem

minimize \[ x_1 + x_2 + x_3 + x_4 \]
subject to
\[
\begin{align*}
-2x_1 + 8x_2 + 0x_3 + 10x_4 & \geq 50 \\
5x_1 + 2x_2 + 0x_3 + 0x_4 & \geq 100 \\
3x_1 - 5x_2 + 10x_3 - 2x_4 & \geq 25 \\
x_1, x_2, x_3, x_4 & \geq 0
\end{align*}
\]

The solution of this linear program yields the optimal advertising strategy.

Formal Definition of Linear Program

- Given \(a_1, a_2, \ldots, a_n\) and a set of variables \(x_1, x_2, \ldots, x_n\), a linear function \(f\) is defined by
  \[
  f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n.
  \]

- **Linear Equality:** \(f(x_1, x_2, \ldots, x_n) = b\) 
- **Linear Inequality:** \(f(x_1, x_2, \ldots, x_n) \geq b\) 
- **Linear-Programming Problem:** either minimize or maximize a linear function subject to a set of linear constraints
A Small(er) Example

maximize \( x_1 + x_2 \)
subject to
\[
\begin{align*}
4x_1 - x_2 &\leq 8 \\
2x_1 + x_2 &\leq 10 \\
5x_1 - 2x_2 &\geq -2 \\
x_1, x_2 &\geq 0
\end{align*}
\]

Any setting of \( x_1 \) and \( x_2 \) satisfying all constraints is a feasible solution.
A Small(er) Example

maximize \[ x_1 + x_2 \]
subject to
\[ 4x_1 - x_2 \leq 8 \]
\[ 2x_1 + x_2 \leq 10 \]
\[ 5x_1 - 2x_2 \geq -2 \]
\[ x_1, x_2 \geq 0 \]

Graphical Procedure: Move the line \( x_1 + x_2 = z \) as far up as possible.

While the same approach also works for higher-dimensions, we need to take a more systematic and algebraic procedure.
Outline

Introduction

Standard and Slack Forms

Formulating Problems as Linear Programs

Simplex Algorithm

Finding an Initial Solution
Standard and Slack Forms

**Standard Form**

maximize  \[ \sum_{j=1}^{n} c_j x_j \]  

subject to  

\[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \]  \hspace{1cm} \text{for } i = 1, 2, \ldots, m 

\[ x_j \geq 0 \]  \hspace{1cm} \text{for } j = 1, 2, \ldots, n

**Standard Form (Matrix-Vector-Notation)**

maximize  \[ c^T x \]  

subject to  

\[ Ax \leq b \] 

\[ x \geq 0 \]
Converting Linear Programs into Standard Form

Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.
2. There might be variables without nonnegativity constraints.
3. There might be equality constraints.
4. There might be inequality constraints (with $\geq$ instead of $\leq$).

Goal: Convert linear program into an equivalent program which is in standard form

Equivalence: a correspondence (not necessarily a bijection) between solutions so that their objective values are identical.

When switching from maximization to minimization, sign of objective value changes.
Reasons for a LP not being in standard form:
1. The objective might be a minimization rather than maximization.

\[
\begin{align*}
\text{minimize} & \quad -2x_1 + 3x_2 \\
\text{subject to} & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \leq 4 \\
& x_1 \geq 0 \\
\end{align*}
\]

Negate objective function

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x_2 \\
\text{subject to} & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \leq 4 \\
& x_1 \geq 0 \\
\end{align*}
\]
Reasons for a LP not being in standard form:

2. There might be variables without nonnegativity constraints.

maximize $2x_1 - 3x_2$
subject to

\[
\begin{align*}
  x_1 + x_2 &= 7 \\
  x_1 - 2x_2 &\leq 4 \\
  x_1 &\geq 0
\end{align*}
\]

Replace $x_2$ by two non-negative variables $x_2'$ and $x_2''$

maximize $2x_1 - 3x_2' + 3x_2''$
subject to

\[
\begin{align*}
  x_1 + x_2' - x_2'' &= 7 \\
  x_1 - 2x_2' + 2x_2'' &\leq 4 \\
  x_1, x_2', x_2'' &\geq 0
\end{align*}
\]
Reasons for a LP not being in standard form:

3. There might be equality constraints.

maximize \[ 2x_1 - 3x'_2 + 3x''_2 \]
subject to
\[ x_1 + x'_2 - x''_2 = 7 \]
\[ x_1 - 2x'_2 + 2x''_2 \leq 4 \]
\[ x_1, x'_2, x''_2 \geq 0 \]

Replace each equality by two inequalities.

maximize \[ 2x_1 - 3x'_2 + 3x''_2 \]
subject to
\[ x_1 + x'_2 - x''_2 \leq 7 \]
\[ x_1 + x'_2 - x''_2 \geq 7 \]
\[ x_1 - 2x'_2 + 2x''_2 \leq 4 \]
\[ x_1, x'_2, x''_2 \geq 0 \]
Converting into Standard Form (4/5)

Reasons for a LP not being in standard form:
4. There might be inequality constraints (with \( \geq \) instead of \( \leq \)).

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x'_2 + 3x''_2 \\
\text{subject to} & \quad x_1 + x'_2 - x''_2 \leq 7 \\
& \quad x_1 + x'_2 - x''_2 \geq 7 \\
& \quad x_1 - 2x'_2 + 2x''_2 \leq 4 \\
& \quad x_1, x'_2, x''_2 \geq 0
\end{align*}
\]

Negate respective inequalities.

\[
\begin{align*}
\text{maximize} & \quad 2x_1 - 3x'_2 + 3x''_2 \\
\text{subject to} & \quad -x_1 - x'_2 + x''_2 \leq -7 \\
& \quad -x_1 - x'_2 + x''_2 \leq -7 \\
& \quad x_1 - 2x'_2 + 2x''_2 \leq 4 \\
& \quad x_1, x'_2, x''_2 \geq 0
\end{align*}
\]
Converting into Standard Form (5/5)

Rewrite the given linear program into standard form.

Maximize: \( 2x_1 - 3x_2 + 3x_3 \)
Subject to:

\[
\begin{align*}
x_1 + x_2 - x_3 & \leq 7 \\
-x_1 - x_2 + x_3 & \leq -7 \\
x_1 - 2x_2 + 2x_3 & \leq 4 \\
x_1, x_2, x_3 & \geq 0
\end{align*}
\]

It is always possible to convert a linear program into standard form.
**Goal:** Convert standard form into slack form, where all constraints except for the non-negativity constraints are equalities.

For the simplex algorithm, it is more convenient to work with equality constraints.

---

Introducing Slack Variables

- Let \( \sum_{j=1}^{n} a_{ij}x_j \leq b_i \) be an inequality constraint
- Introduce a slack variable \( s \) by

\[
s = b_i - \sum_{j=1}^{n} a_{ij}x_j
\]

\( s \geq 0 \).

- Denote slack variable of the \( i \)th inequality by \( x_{n+i} \)

\( s \) measures the slack between the two sides of the inequality.
Converting Standard Form into Slack Form (2/3)

maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[
\begin{align*}
  x_1 &+ x_2 - x_3 &\leq & 7 \\
-x_1 &- x_2 &+ x_3 &\leq -7 \\
  x_1 &- 2x_2 &+ 2x_3 &\leq 4 \\
  x_1, x_2, x_3 &\geq & 0 \\
\end{align*}
\]

Introduce slack variables

maximize \[ 2x_1 - 3x_2 + 3x_3 \]
subject to
\[
\begin{align*}
  x_4 &= 7 - x_1 - x_2 + x_3 \\
  x_5 &= -7 + x_1 + x_2 - x_3 \\
  x_6 &= 4 - x_1 + 2x_2 - 2x_3 \\
  x_1, x_2, x_3, x_4, x_5, x_6 &\geq 0 \\
\end{align*}
\]
maximize
subject to

\[ \begin{align*}
2x_1 & - 3x_2 + 3x_3 \\
\end{align*} \]

\[ \begin{align*}
x_4 & = 7 - x_1 - x_2 + x_3 \\
x_5 & = -7 + x_1 + x_2 - x_3 \\
x_6 & = 4 - x_1 + 2x_2 - 2x_3 \\
\end{align*} \]

\[ \begin{align*}
x_1, x_2, x_3, x_4, x_5, x_6 & \geq 0 \\
\end{align*} \]

Use variable \( z \) to denote objective function and omit the nonnegativity constraints.

\[ \begin{align*}
z & = 2x_1 - 3x_2 + 3x_3 \\
x_4 & = 7 - x_1 - x_2 + x_3 \\
x_5 & = -7 + x_1 + x_2 - x_3 \\
x_6 & = 4 - x_1 + 2x_2 - 2x_3 \\
\end{align*} \]

This is called **slack form**.
Basic and Non-Basic Variables

\[ z = 2x_1 - 3x_2 + 3x_3 \]
\[ x_4 = 7 - x_1 - x_2 + x_3 \]
\[ x_5 = -7 + x_1 + x_2 - x_3 \]
\[ x_6 = 4 - x_1 + 2x_2 - 2x_3 \]

Basic Variables: \( B = \{4, 5, 6\} \)

Non-Basic Variables: \( N = \{1, 2, 3\} \)

Slack Form (Formal Definition)

Slack form is given by a tuple \((N, B, A, b, c, v)\) so that

\[ z = v + \sum_{j \in N} c_j x_j \]
\[ x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B, \]

and all variables are non-negative.

Variables on the right hand side are indexed by the entries of \( N \).
Slack Form (Example)

\[
\begin{align*}
  z & = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
  x_1 & = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
  x_2 & = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
  x_4 & = 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{align*}
\]

---

**Slack Form Notation**

- \( B = \{1, 2, 4\}, \ N = \{3, 5, 6\} \)
- \[
  A = \begin{pmatrix}
    a_{13} & a_{15} & a_{16} \\
    a_{23} & a_{25} & a_{26} \\
    a_{43} & a_{45} & a_{46}
  \end{pmatrix} = \begin{pmatrix}
    -1/6 & -1/6 & 1/3 \\
    8/3 & 2/3 & -1/3 \\
    1/2 & -1/2 & 0
  \end{pmatrix}
\]
- \[
  b = \begin{pmatrix}
    b_1 \\
    b_2 \\
    b_3
  \end{pmatrix} = \begin{pmatrix}
    8 \\
    4 \\
    18
  \end{pmatrix}, \quad
  c = \begin{pmatrix}
    c_3 \\
    c_5 \\
    c_6
  \end{pmatrix} = \begin{pmatrix}
    -1/6 \\
    -1/6 \\
    -2/3
  \end{pmatrix}
\]
- \( v = 28 \)
The Structure of Optimal Solutions

Definition

A point \( x \) is a vertex if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

Theorem

If there exists an optimal solution, one of them occurs at a vertex.

Proof:

- Let \( x \) be an optimal solution which is not a vertex
  \( \Rightarrow \exists \) vector \( d \) so that \( x - d \) and \( x + d \) are feasible
- Since \( A(x + d) = b \) and \( Ax = b \Rightarrow Ad = 0 \)
- W.l.o.g. assume \( c^T d \geq 0 \) (otherwise replace \( d \) by \( -d \))
- Consider \( x + \lambda d \) as a function of \( \lambda \geq 0 \)

Case 1: There exists \( j \) with \( d_j < 0 \)

- Increase \( \lambda \) from 0 to \( \lambda' \) until a new entry of \( x + \lambda d \) becomes zero
- \( x + \lambda' d \) feasible, since \( A(x + \lambda' d) = Ax = b \) and \( x + \lambda' d \geq 0 \)
- \( c^T (x + \lambda' d) = c^T x + c^T \lambda' d \geq c^T x \)
The Structure of Optimal Solutions

**Definition**
A point $x$ is a **vertex** if it cannot be represented as a strict convex combination of two other points in the feasible set.

The set of feasible solutions is a convex set.

**Theorem**
If there exists an optimal solution, one of them occurs at a vertex.

**Proof:**
- Let $x$ be an optimal solution which is not a vertex
  $\Rightarrow \exists$ vector $d$ so that $x - d$ and $x + d$ are feasible
- Since $A(x + d) = b$ and $Ax = b \Rightarrow Ad = 0$
- W.l.o.g. assume $c^T d \geq 0$ (otherwise replace $d$ by $-d$)
- Consider $x + \lambda d$ as a function of $\lambda \geq 0$

**Case 2:** For all $j$, $d_j \geq 0$
- $x + \lambda d$ is feasible for all $\lambda \geq 0$: $A(x + \lambda d) = b$ and $x + \lambda d \geq x \geq 0$
- If $\lambda \to \infty$, then $c^T (x + \lambda d) \to \infty$
  $\Rightarrow$ This contradicts the assumption that there exists an optimal solution.
Outline

Introduction

Standard and Slack Forms

Formulating Problems as Linear Programs

Simplex Algorithm

Finding an Initial Solution
Shortest Paths

**Single-Pair Shortest Path Problem**

- **Given:** directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$, pair of vertices $s, t \in V$
- **Goal:** Find a path of minimum weight from $s$ to $t$ in $G$

$p = (v_0 = s, v_1, \ldots, v_k = t)$ such that $w(p) = \sum_{i=1}^{k} w(v_{k-1}, v_k)$ is minimized.

**Shortest Paths as LP**

- **Maximize:** $d_t$
- **Subject to:**
  - $d_v \leq d_u + w(u, v)$ for each edge $(u, v) \in E$,
  - $d_s = 0$.

Recall: When BELLMAN-FORD terminates, all these inequalities are satisfied.

Solution $\bar{d}$ satisfies $\bar{d}_v = \min_{u: (u,v) \in E} \left\{ \bar{d}_u + w(u, v) \right\}$
Maximum Flow

**Maximum Flow Problem**

- **Given**: directed graph $G = (V, E)$ with edge capacities $c : E \to \mathbb{R}^+$, pair of vertices $s, t \in V$
- **Goal**: Find a maximum flow $f : V \times V \to \mathbb{R}$ from $s$ to $t$ which satisfies the capacity constraints and flow conservation

**Maximum Flow as LP**

$maximize \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs}$

subject to

- $f_{uv} \leq c(u, v)$ for each $u, v \in V$,
- $\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv}$ for each $u \in V \setminus \{s, t\}$,
- $f_{uv} \geq 0$ for each $u, v \in V$. 

**Example Graph**

![Graph example](image)
Minimum-Cost Flow

Minimum-Cost-Flow Problem

- Given: directed graph $G = (V, E)$ with capacities $c : E \rightarrow \mathbb{R}^+$, pair of vertices $s, t \in V$, cost function $a : E \rightarrow \mathbb{R}^+$, flow demand of $d$ units
- Goal: Find a flow $f : V \times V \rightarrow \mathbb{R}$ from $s$ to $t$ with $|f| = d$ while minimising the total cost $\sum_{(u,v) \in E} a(u, v) f_{uv}$ incurred by the flow.

Optimal Solution with total cost:
$\sum_{(u,v) \in E} a(u, v) f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$

Figure 29.3  (a) An example of a minimum-cost-flow problem. We denote the capacities by $c$ and the costs by $a$. Vertex $s$ is the source and vertex $t$ is the sink, and we wish to send 4 units of flow from $s$ to $t$. (b) A solution to the minimum-cost flow problem in which 4 units of flow are sent from $s$ to $t$. For each edge, the flow and capacity are written as flow/capacity.
Minimum Cost Flow as LP

minimize \[ \sum_{(u,v) \in E} a(u,v) f_{uv} \]
subject to
\[ f_{uv} \leq c(u,v) \quad \text{for each } u, v \in V, \]
\[ \sum_{v \in V} f_{vu} - \sum_{v \in V} f_{uv} = 0 \quad \text{for each } u \in V \setminus \{s, t\}, \]
\[ \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} = d, \]
\[ f_{uv} \geq 0 \quad \text{for each } u, v \in V. \]

Real power of Linear Programming comes from the ability to solve new problems!
Outline

Introduction

Standard and Slack Forms

Formulating Problems as Linear Programs

Simplex Algorithm

Finding an Initial Solution
Simplex Algorithm: Introduction

Simplex Algorithm

- classical method for solving linear programs (Dantzig, 1947)
- usually fast in practice although worst-case runtime not polynomial
- iterative procedure somewhat similar to Gaussian elimination

Basic Idea:

- Each iteration corresponds to a “basic solution” of the slack form
- All non-basic variables are 0, and the basic variables are determined from the equality constraints
- Each iteration converts one slack form into an equivalent one while the objective value will not decrease
- Conversion (“pivoting”) is achieved by switching the roles of one basic and one non-basic variable

In that sense, it is a greedy algorithm.
Extended Example: Conversion into Slack Form

maximize \[ 3x_1 + x_2 + 2x_3 \]
subject to
\[ x_1 + x_2 + 3x_3 \leq 30 \]
\[ 2x_1 + 2x_2 + 5x_3 \leq 24 \]
\[ 4x_1 + x_2 + 2x_3 \leq 36 \]
\[ x_1, x_2, x_3 \geq 0 \]

Conversion into slack form

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]
Extended Example: Iteration 1

Basic solution: 

\[(x_1, x_2, \ldots, x_6) = (0, 0, 0, 30, 24, 36)\]

This basic solution is feasible. Objective value is 0.

The third constraint is the tightest and limits how much we can increase \(x_1\).

The third constraint is the tightest and limits how much we can increase \(x_3\).

The second constraint is the tightest and limits how much we can increase \(x_2\).
Extended Example: Iteration 1

Increasing the value of $x_1$ would increase the objective value.

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

The third constraint is the tightest and limits how much we can increase $x_1$.

Switch roles of $x_1$ and $x_6$:
- Solving for $x_1$ yields:
  \[ x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}. \]
- Substitute this into $x_1$ in the other three equations

III. Linear Programming
Simplex Algorithm
Extended Example: Iteration 2

Increasing the value of $x_3$ would increase the objective value.

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4}$$
$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}$$
$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4}$$
$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}$$

Basic solution: $(x_1, x_2, \ldots, x_6) = (9, 0, 0, 21, 6, 0)$ with objective value 27
Extended Example: Iteration 2

\[
\begin{align*}
    z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
    x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
    x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
    x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{align*}
\]

The third constraint is the tightest and limits how much we can increase \( x_3 \).

**Switch roles of \( x_3 \) and \( x_5 \):**

- Solving for \( x_3 \) yields:
  \[
  x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} - \frac{x_6}{8}.
  \]

- Substitute this into \( x_3 \) in the other three equations
Extended Example: Iteration 3

Increasing the value of $x_2$ would increase the objective value.

\[
\begin{align*}
    z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

Basic solution: \((x_1, x_2, \ldots, x_6) = (\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0)\) with objective value \(\frac{111}{4} = 27.75\)
Extended Example: Iteration 3

\[
\begin{align*}
    z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
    x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
    x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
    x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

The second constraint is the tightest and limits how much we can increase \(x_2\).

**Switch roles of \(x_2\) and \(x_3\):**
- Solving for \(x_2\) yields:
  \[
  x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}.
  \]
- Substitute this into \(x_2\) in the other three equations.
Extended Example: Iteration 4

All coefficients are negative, and hence this basic solution is optimal!

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]

Basic solution: \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (8, 4, 0, 18, 0, 0)\) with objective value 28
Extended Example: Visualization of SIMPLEX

(0, 0, 4.8)  

9.6  

(0, 0, 0)  

0  

(8.25, 0, 1.5)  

27.75  

(8, 4, 0)  

28  

(9, 0, 0)  

27  

III. Linear Programming  
Simplex Algorithm
Extended Example: Alternative Runs (1/2)

\[ z = 3x_1 + x_2 + 2x_3 \]
\[ x_4 = 30 - x_1 - x_2 - 3x_3 \]
\[ x_5 = 24 - 2x_1 - 2x_2 - 5x_3 \]
\[ x_6 = 36 - 4x_1 - x_2 - 2x_3 \]

Switch roles of \(x_2\) and \(x_5\)

\[ z = 12 + 2x_1 - \frac{x_3}{2} - \frac{x_5}{2} \]
\[ x_2 = 12 - x_1 - \frac{5x_3}{2} - \frac{x_5}{2} \]
\[ x_4 = 18 - x_2 - \frac{x_3}{2} + \frac{x_5}{2} \]
\[ x_6 = 24 - 3x_1 + \frac{x_3}{2} + \frac{x_5}{2} \]

Switch roles of \(x_1\) and \(x_6\)

\[ z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \]
\[ x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \]
\[ x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \]
\[ x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} \]
Extended Example: Alternative Runs (2/2)

\[
\begin{align*}
z &= 3x_1 + x_2 + 2x_3 \\
x_4 &= 30 - x_1 - x_2 - 3x_3 \\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\
x_6 &= 36 - 4x_1 - x_2 - 2x_3
\end{align*}
\]

Switch roles of \(x_3\) and \(x_5\)

\[
\begin{align*}
z &= \frac{48}{5} + \frac{11x_1}{5} + \frac{x_2}{5} - \frac{2x_5}{5} \\
x_4 &= \frac{78}{5} + \frac{x_1}{5} + \frac{x_2}{5} + \frac{3x_5}{5} \\
x_3 &= \frac{24}{5} - \frac{2x_1}{5} - \frac{2x_2}{5} - \frac{x_5}{5} \\
x_6 &= \frac{132}{5} - \frac{16x_1}{5} - \frac{x_2}{5} + \frac{2x_3}{5}
\end{align*}
\]

Switch roles of \(x_1\) and \(x_6\)  
Switch roles of \(x_2\) and \(x_3\)

\[
\begin{align*}
\frac{z}{4} &= 11 + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
x_1 &= 33 + \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
x_4 &= 69 + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{align*}
\]

Switch roles of \(x_3\) and \(x_5\)

\[
\begin{align*}
\frac{z}{4} &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{align*}
\]

III. Linear Programming  
Simplex Algorithm  
33
The Pivot Step Formally

\textbf{Pivot}(N, B, A, b, c, v, l, e)

1. \hspace{1em} // Compute the coefficients of the equation for new basic variable \( x_e \).
2. \hspace{1em} let \( \hat{A} \) be a new \( m \times n \) matrix
3. \hspace{1em} \( \hat{b}_e = b_l / a_{le} \)
4. \hspace{1em} for each \( j \in N - \{e\} \)
5. \hspace{1.2em} \( \hat{a}_{ej} = a_{lj} / a_{le} \)
6. \hspace{1em} \( \hat{a}_{el} = 1 / a_{le} \)
7. \hspace{1em} // Compute the coefficients of the remaining constraints.
8. \hspace{1em} for each \( i \in B - \{l\} \)
9. \hspace{1.2em} \( \hat{b}_i = b_i - a_{ie} \hat{b}_e \)
10. \hspace{1.2em} for each \( j \in N - \{e\} \)
11. \hspace{1.4em} \( \hat{a}_{ij} = a_{ij} - a_{ie} \hat{a}_{ej} \)
12. \hspace{1.4em} \( \hat{a}_{il} = -a_{ie} \hat{a}_{el} \)
13. \hspace{1em} // Compute the objective function.
14. \hspace{1em} \( \hat{v} = v + c_e \hat{b}_e \)
15. \hspace{1em} for each \( j \in N - \{e\} \)
16. \hspace{1.2em} \( \hat{c}_j = c_j - c_e \hat{a}_{ej} \)
17. \hspace{1.2em} \( \hat{c}_l = -c_e \hat{a}_{el} \)
18. \hspace{1em} // Compute new sets of basic and nonbasic variables.
19. \hspace{1em} \( \hat{N} = N - \{e\} \cup \{l\} \)
20. \hspace{1em} \( \hat{B} = B - \{l\} \cup \{e\} \)
21. \hspace{1em} return \( (\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v}) \)

Rewrite “tight” equation for entering variable \( x_e \).

Substituting \( x_e \) into other equations.

Substituting \( x_e \) into objective function.

Update non-basic and basic variables.
Effect of the Pivot Step

Lemma 29.1

Consider a call to $\text{PIVOT}(N, B, A, b, c, v, l, e)$ in which $a_{le} \neq 0$. Let the values returned from the call be $(\hat{N}, \hat{B}, \hat{A}, \hat{b}, \hat{c}, \hat{v})$, and let $\bar{x}$ denote the basic solution after the call. Then

1. $\bar{x}_j = 0$ for each $j \in \hat{N}$.
2. $\bar{x}_e = b_l / a_{le}$.
3. $\bar{x}_i = b_i - a_{ie}\hat{b}_e$ for each $i \in \hat{B} \setminus \{e\}$.

Proof:

1. holds since the basic solution always sets all non-basic variables to zero.
2. When we set each non-basic variable to 0 in a constraint

   $$x_i = \hat{b}_i - \sum_{j \in \hat{N}} \hat{a}_{ij}x_j,$$

   we have $\bar{x}_i = \hat{b}_i$ for each $i \in \hat{B}$. Hence $\bar{x}_e = \hat{b}_e = b_l / a_{le}$.
3. After the substituting in the other constraints, we have

   $$\bar{x}_i = \hat{b}_i = b_i - a_{ie}\hat{b}_e.$$ 

III. Linear Programming

Simplex Algorithm

35
Formalizing the Simplex Algorithm: Questions

Questions:
- How do we determine whether a linear program is feasible?
- What do we do if the linear program is feasible, but the initial basic solution is not feasible?
- How do we determine whether a linear program is unbounded?
- How do we choose the entering and leaving variables?

Example before was a particularly nice one!
In Section 29.5, we shall show how to determine whether a problem is feasible, and if so, how to find a slack form in which the initial basic solution is feasible. Therefore, let us assume that we have a procedure $I$ that takes as input a linear program in standard form, that is, an equation described in (29.19)–(29.21). It returns an $m$-vector $D$ that is positive, as the entering variable. Although we may choose any such variable as the entering variable, we assume that we use some prespecified deterministic rule.

The procedure $S$ described above, which determines whether the program is feasible or returns a slack form for which the initial basic solution is feasible, terminates if all coefficients in objective function are negative. Otherwise, line 4 selects a variable that takes as input a linear program in standard form, that is, an equation described. It returns an $m$-vector $D$, that minimizes $\Delta_i$, $\forall i \in N$. If $\Delta_i = \infty$, choose an index $l \in B$ that minimizes $\Delta_i$. If $\Delta_l = \infty$, return “unbounded”. Otherwise, line 12 calls $P$ with negative coefficient $x_e$ with negative coefficient, associated with $x_i$. Line 11 returns “unbounded” if there are no constraints. Line 12 calls $P$ with negative coefficient, switching roles of $x_i$ and $x_e$.
The formal procedure SIMPLEX

SIMPLEX\((A, b, c)\)

1. \((N, B, A, b, c, ν) = \text{INITIALIZE-SIMPLEX}(A, b, c)\)
2. let \(Δ\) be a new vector of length \(n\)
3. while some index \(j ∈ N\) has \(c_j > 0\)
4. choose an index \(e ∈ N\) for which \(c_e > 0\)
5. for each index \(i ∈ B\)
6. if \(a_{ie} > 0\)
7. \(Δ_i = b_i / a_{ie}\)
8. else \(Δ_i = ∞\)
9. choose an index \(l ∈ B\) that minimizes \(Δ_i\)
10. if \(Δ_l = ∞\)
11. return “unbounded”

**Proof** is based on the following three-part loop invariant:

1. the slack form is always equivalent to the one returned by \text{INITIALIZE-SIMPLEX},
2. for each \(i ∈ B\), we have \(b_i ≥ 0\),
3. the basic solution associated with the (current) slack form is feasible.

---

**Lemma 29.2**

Suppose the call to \text{INITIALIZE-SIMPLEX} in line 1 returns a slack form for which the basic solution is feasible. Then if SIMPLEX returns a solution, it is a feasible solution. If SIMPLEX returns “unbounded”, the linear program is unbounded.
**Termination**

**Degeneracy**: One iteration of SIMPLEX leaves the objective value unchanged.

\[
egin{align*}
 z &= x_1 + x_2 + x_3 \\
 x_4 &= 8 - x_1 - x_2 \\
 x_5 &= x_2 - x_3
\end{align*}
\]

Pivot with \(x_1\) entering and \(x_4\) leaving

\[
egin{align*}
 z &= 8 + x_3 - x_4 \\
 x_1 &= 8 - x_2 - x_4 \\
 x_5 &= x_2 - x_3
\end{align*}
\]

Cycling: Slack forms at two iterations are identical, and SIMPLEX fails to terminate!

\[
egin{align*}
 z &= 8 + x_2 - x_4 - x_5 \\
 x_1 &= 8 - x_2 - x_4 \\
 x_3 &= x_2 - x_5
\end{align*}
\]

III. Linear Programming

Simplex Algorithm
Termination and Running Time

**Cycling**: SIMPLEX may fail to terminate.

### Anti-Cycling Strategies

1. **Bland’s rule**: Choose entering variable with smallest index
2. **Random rule**: Choose entering variable uniformly at random
3. **Perturbation**: Perturb the input slightly so that it is impossible to have two solutions with the same objective value

Replace each $b_i$ by $\hat{b}_i = b_i + \epsilon_i$, where $\epsilon_i \gg \epsilon_{i+1}$ are all small.

**Lemma 29.7**

Assuming `INITIALIZE-SIMPLEX` returns a slack form for which the basic solution is feasible, SIMPLEX either reports that the program is unbounded or returns a feasible solution in at most $\binom{n+m}{m}$ iterations.

Every set $B$ of basic variables uniquely determines a slack form, and there are at most $\binom{n+m}{m}$ unique slack forms.
Outline

Introduction

Standard and Slack Forms

Formulating Problems as Linear Programs

Simplex Algorithm

Finding an Initial Solution
Finding an Initial Solution

maximize \[ 2x_1 - x_2 \]
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
\end{align*}
\]
\[
\begin{align*}
x_1, x_2 & \geq 0 \\
\end{align*}
\]

Conversion into slack form

\[
\begin{align*}
z &= 2x_1 - x_2 \\
x_3 &= 2 - 2x_1 + x_2 \\
x_4 &= -4 - x_1 + 5x_2 \\
\end{align*}
\]

Basic solution \((x_1, x_2, x_3, x_4) = (0, 0, 2, -4)\) is not feasible!
maximize \( 2x_1 - x_2 \)
subject to
\[
\begin{align*}
2x_1 - x_2 & \leq 2 \\
x_1 - 5x_2 & \leq -4 \\
x_1, x_2 & \geq 0 \\
\end{align*}
\]

Questions:
- How to determine whether there is any feasible solution?
- If there is one, how to determine an initial basic solution?
Formulating an Auxiliary Linear Program

maximize \[ \sum_{j=1}^{n} c_j x_j \]
subject to
\[ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m, \]
\[ x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n \]

Let \( L_{aux} \) be the auxiliary LP of a linear program \( L \) in standard form. Then \( L \) is feasible if and only if the optimal objective value of \( L_{aux} \) is 0.

**Proof.**

- “\( \Rightarrow \)”: Suppose \( L \) has a feasible solution \( \bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n) \)
  - \( \bar{x}_0 = 0 \) combined with \( \bar{x} \) is a feasible solution to \( L_{aux} \) with objective value 0.
  - Since \( \bar{x}_0 \geq 0 \) and the objective is to maximize \(-\bar{x}_0\), this is optimal for \( L_{aux} \)
- “\( \Leftarrow \)”: Suppose that the optimal objective value of \( L_{aux} \) is 0
  - Then \( \bar{x}_0 = 0 \), and the remaining solution values \((\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)\) satisfy \( L \). \qed
### INITIALIZE-SIMPLEX

**INITIALIZE-SIMPLEX** \((A, b, c)\)

1. let \(k\) be the index of the minimum \(b_i\)
2. if \(b_k \geq 0\)  
   // is the initial basic solution feasible?
   return \(\{1, 2, \ldots, n\}, \{n + 1, n + 2, \ldots, n + m\}, A, b, c, 0\)
3. form \(L_{aux}\) by adding \(-x_0\) to the left-hand side of each constraint 
   and setting the objective function to \(-x_0\)
4. let \((N, B, A, b, c, v)\) be the resulting slack form for \(L_{aux}\)
5. \(l = n + k\)
6. // \(L_{aux}\) has \(n + 1\) nonbasic variables and \(m\) basic variables.
7. \((N, B, A, b, c, v) = \text{PIVOT}(N, B, A, b, c, v, l, 0)\)
8. // The basic solution is now feasible for \(L_{aux}\).
9. iterate the while loop of lines 3–12 of SIMPLEX until an optimal solution 
   to \(L_{aux}\) is found
10. if the optimal solution to \(L_{aux}\) sets \(x_0\) to 0
11.   if \(x_0\) is basic
12.     perform one (degenerate) pivot to make it nonbasic
13.     from the final slack form of \(L_{aux}\), remove \(x_0\) from the constraints and 
     restore the original objective function of \(L\), but replace each basic 
     variable in this objective function by the right-hand side of its 
     associated constraint
14. return the modified final slack form
15. else return “infeasible”

---

Test solution with \(N = \{1, 2, \ldots, n\}, B = \{n + 1, n + 2, \ldots, n + m\}\), \(\bar{x}_i = b_i\) for \(i \in B\), \(\bar{x}_i = 0\) otherwise.

\(\ell\) will be the leaving variable so that \(x_\ell\) has the most negative value.

Pivot step with \(x_\ell\) leaving and \(x_0\) entering.

This pivot step does not change the value of any variable.
Example of **INITIALIZE-SIMPLEX (1/3)**

maximize \[ 2x_1 - x_2 \]
subject to
\[ 2x_1 - x_2 \leq 2 \]
\[ x_1 - 5x_2 \leq -4 \]
\[ x_1, x_2 \geq 0 \]

Formulating the auxiliary linear program

maximize \[ -x_0 \]
subject to
\[ 2x_1 - x_2 - x_0 \leq 2 \]
\[ x_1 - 5x_2 - x_0 \leq -4 \]
\[ x_1, x_2, x_0 \geq 0 \]

Basic solution \((0, 0, 0, 2, -4)\) not feasible!

Converting into slack form

\[
\begin{align*}
z &= 2 - 2x_1 + x_2 + x_0 \\
x_3 &= 2 - 2x_1 + x_2 + x_0 \\
x_4 &= -4 - x_1 + 5x_2 + x_0
\end{align*}
\]
Example of INITIALIZE-SIMPLEX (2/3)

\[
\begin{align*}
  z &= 2x_1 + x_2 - x_0 \\
  x_3 &= 2x_1 + x_2 + x_0 \\
  x_4 &= -4x_1 + 5x_2 + x_0 \\
\end{align*}
\]

Pivot with \( x_0 \) entering and \( x_4 \) leaving

\[
\begin{align*}
  z &= -4x_1 + 5x_2 - x_4 \\
  x_0 &= 4x_1 - 5x_2 + x_4 \\
  x_3 &= 6x_1 - 4x_2 + x_4 \\
\end{align*}
\]

Basic solution \((4, 0, 0, 6, 0)\) is feasible!

Pivot with \( x_2 \) entering and \( x_0 \) leaving

\[
\begin{align*}
  z &= 4x_1 - x_0 \\
  x_2 &= \frac{4}{5}x_1 - \frac{1}{5}x_0 + \frac{9}{5}x_1 + \frac{4}{5}x_4 \\
  x_3 &= \frac{14}{5} + \frac{4}{5}x_0 - \frac{9}{5}x_1 + \frac{4}{5}x_4 \\
\end{align*}
\]

Optimal solution has \( x_0 = 0 \), hence the initial problem was feasible!
Example of INITIALIZE-SIMPLEX (3/3)

\[ z = -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \]
\[ x_2 = \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \]
\[ x_3 = \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \]

Set \( x_0 = 0 \) and express objective function by non-basic variables

Basic solution \((0, \frac{4}{5}, \frac{14}{5}, 0)\), which is feasible!

Lemma 29.12

If a linear program \( L \) has no feasible solution, then INITIALIZE-SIMPLEX returns “infeasible”. Otherwise, it returns a valid slack form for which the basic solution is feasible.
Fundamental Theorem of Linear Programming

Theorem 29.13

Any linear program \( L \), given in standard form, either

1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded.

If \( L \) is infeasible, \textsc{Simplex} returns “infeasible”. If \( L \) is unbounded, \textsc{Simplex} returns “unbounded”. Otherwise, \textsc{Simplex} returns an optimal solution with a finite objective value.
Linear Programming and Simplex: Summary

**Linear Programming**
- extremely versatile tool for modelling problems of all kinds
- basis of Integer Programming, to be discussed in later lectures

**Simplex Algorithm**
- In practice: usually terminates in polynomial time, i.e., $O(m + n)$
- In theory: even with anti-cycling may need exponential time

**Research Problem**: Is there a pivoting rule which makes SIMPLEX a polynomial-time algorithm?

**Polynomial-Time Algorithms**
- Interior-Point Methods: traverses the interior of the feasible set of solutions (not just vertices!)
IV. Approximation Algorithms: Covering Problems

Thomas Sauerwald
Introduction

Vertex Cover

The Set-Covering Problem
Motivation

Many fundamental problems are **NP-complete**, yet they are too important to be abandoned.

Examples: HAMILTON, 3-SAT, VERTEX-COVER, KNAPSACK, ...

**Strategies to cope with NP-complete problems**

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

We will call these **approximation algorithms**.
An algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

This covers both maximization and minimization problems.

For many problems: tradeoff between runtime and approximation ratio.

**Approximation Schemes**

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^2/\epsilon)$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. For example, $O((1/\epsilon)^2 \cdot n^3)$. 
Outline

Introduction

Vertex Cover

The Set-Covering Problem
The Vertex-Cover Problem

- We are covering edges by picking vertices!

Vertex Cover Problem

- **Given:** Undirected graph $G = (V, E)$
- **Goal:** Find a minimum-cardinality subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is an NP-hard problem.

Applications:

- Every edge forms a task, and every vertex represents a person/machine which can execute that task
- Perform all tasks with the minimal amount of resources
- Extensions: weighted edges or hypergraphs
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER**(*G*)

1. \( C = \emptyset \)
2. \( E' = G.E \)
3. \( \textbf{while } E' \neq \emptyset \)
4. \( \text{let } (u, v) \text{ be an arbitrary edge of } E' \)
5. \( C = C \cup \{u, v\} \)
6. \( \text{remove from } E' \text{ every edge incident on either } u \text{ or } v \)
7. \( \textbf{return } C \)

Figure 35.1 illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \( C \) contains the vertex cover being constructed. Line 1 initializes \( C \) to the empty set. Line 2 sets \( E' \) to be a copy of the edge set \( G.E \) of the graph. The loop of lines 3–6 repeatedly picks an edge \( (u, v) \) from \( E' \), adds it to \( C \), and removes from \( E' \) every edge incident on either \( u \) or \( v \). Line 7 returns the computed vertex cover.
An Approximation Algorithm based on Greedy

**APPROX-VERTEX-COVER** ($G$)

1. $C = \emptyset$
2. $E' = G.E$
3. **while** $E' \neq \emptyset$
   4. let $(u, v)$ be an arbitrary edge of $E'$
   5. $C = C \cup \{u, v\}$
   6. remove from $E'$ every edge incident on either $u$ or $v$
4. return $C$

**Figure 35.1** illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable $C$ contains the vertex cover being constructed. Line 1 initializes $C$ to the empty set. Line 2 sets $E'$ to be a copy of the edge set $G.E$ of the graph. The loop of lines 3–6 repeatedly picks an edge $(u, v)$ from $E'$, adds it to $C$, and removes from $E'$ every edge incident on either $u$ or $v$.

**Edges removed from $E'$**:
1. $\{b, c\}$
2. $\{e, f\}$
3. $\{d, g\}$

**APPROX-VERTEX-COVER** produces a set of size 6.

The optimal solution has size 3.
**Approximation Algorithm based on Greedy**

**APPROX-VERTEX-COVER** \((G)\)

1. \(C = \emptyset\)
2. \(E' = G\cdot E\)
3. **while** \(E' \neq \emptyset\)
4. \(\text{let } (u, v) \text{ be an arbitrary edge of } E'\)
5. \(C = C \cup \{u, v\}\)
6. remove from \(E'\) every edge incident on either \(u\) or \(v\)
7. **return** \(C\)

---

Figure 35.1 illustrates how **APPROX-VERTEX-COVER** operates on an example graph. The variable \(C\) contains the vertex cover being constructed. Line 1 initializes \(C\) to the empty set. Line 2 sets \(E'\) to be a copy of the edge set of the graph. The loop of lines 3–6 repeatedly picks an edge \((u, v)\) from \(E'\), adds it to \(C\), and removes from \(E'\) every edge incident on either \(u\) or \(v\). Line 7 returns the vertex cover set \(C\).

The optimal solution has size 3.
Analysis of Greedy for Vertex Cover

**APPROX-VERTEX-COVER**($G$)

1. $C = \emptyset$
2. $E' = G.E$
3. **while** $E' \neq \emptyset$
   4. let $(u, v)$ be an arbitrary edge of $E'$
   5. $C = C \cup \{u, v\}$
   6. remove from $E'$ every edge incident on either $u$ or $v$
4. return $C$

**Theorem 35.1**

**APPROX-VERTEX-COVER** is a poly-time 2-approximation algorithm.

**Proof:**

- **Running time** is $O(V + E)$ (using adjacency lists to represent $E'$)
- Let $A \subseteq E$ denote the set of edges picked in line 4
- Every optimal cover $C^*$ must include at least one endpoint of edges in $A$, and edges in $A$ do not share a common endpoint: $|C^*| \geq |A|$
- Every edge in $A$ contributes 2 vertices to $|C|$: $|C| = 2|A| \leq 2|C^*|$.\hfill\(\blacksquare\)
Solving Special Cases

Strategies to cope with NP-complete problems

1. If inputs are small, an algorithm with exponential running time may be satisfactory.
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.
There exists an **optimal vertex cover** which does not include any **leaves**.

**Exchange-Argument**: Replace any leaf in the cover by its parent.
There exists an optimal vertex cover which does not include any leaves.

\[
\text{VERTEX-COVER-TREES}(G)
\]
1: \( C = \emptyset \)
2: while \( \exists \) leaves in \( G \)
3: Add all parents to \( C \)
4: Remove all leaves and their parents from \( G \)
5: return \( C \)

Clear: Running time is \( O(V) \), and the returned solution is a vertex cover.

Solution is also optimal. (Use inductively the existence of an optimal vertex cover without leaves)
Execution on a Small Example

\textsc{Vertex-Cover-Trees}(G)

1: \( C = \emptyset \)
2: \textbf{while} \( \exists \) leaves in \( G \)
3: \hspace{1em} Add all parents to \( C \)
4: \hspace{1em} Remove all leaves and their parents from \( G \)
5: \textbf{return} \( C \)
Execution on a Small Example

After iteration 1

VERTEX-COVER-TREES(G)
1: \( C = \emptyset \)
2: while \( \exists \) leaves in \( G \)
3: Add all parents to \( C \)
4: Remove all leaves and their parents from \( G \)
5: return \( C \)
Execution on a Small Example

Problem can be also solved on bipartite graphs, using Max-Flows and Min-Cuts.

**VERTEX-COVER-TREES(G)**

1. \( C = \emptyset \)
2. **while** \( \exists \) leaves in \( G \)
3. Add all parents to \( C \)
4. Remove all leaves and their parents from \( G \)
5. **return** \( C \)
Exact Algorithms

Strategies to cope with NP-complete problems

1. If inputs (or solutions) are small, an algorithm with exponential running time may be satisfactory
2. Isolate important special cases which can be solved in polynomial-time.
3. Develop algorithms which find near-optimal solutions in polynomial-time.

Focus on instances of where the minimum vertex cover is small, that is, less or equal than some given integer $k$.

Simple Brute-Force Search would take $\approx \binom{n}{k} = \Theta(n^k)$ time.
Towards a more efficient Search

Substructure Lemma

Consider a graph $G = (V, E)$, edge $(u, v) \in E(G)$ and integer $k \geq 1$. Let $G_u$ be the graph obtained by deleting $u$ and its incident edges ($G_v$ is defined similarly). Then $G$ has a vertex cover of size $k$ if and only if $G_u$ or $G_v$ (or both) have a vertex cover of size $k - 1$.

Proof:

$\Leftarrow$ Assume $G_u$ has a vertex cover $C_u$ of size $k - 1$. Adding $u$ yields a vertex cover of $G$ which is of size $k$.

$\Rightarrow$ Assume $G$ has a vertex cover $C$ of size $k$, which contains, say $u$. Removing $u$ from $C$ yields a vertex cover of $G_u$ which is of size $k - 1$.  

Proof:

Reminiscent of Dynamic Programming.
A More Efficient Search Algorithm

**Vertex-Cover-Search**($G, k$)

1. If $E = \emptyset$ return $\{\bot\}$
2. If $k = 0$ and $E \neq \emptyset$ return $\emptyset$
3. Pick an arbitrary edge $(u, v) \in E$
4. $S_1 = \text{Vertex-Cover-Search}(G_u, k - 1)$
5. $S_2 = \text{Vertex-Cover-Search}(G_v, k - 1)$
6. If $S_1 \neq \emptyset$ return $S_1 \cup \{u\}$
7. If $S_2 \neq \emptyset$ return $S_2 \cup \{v\}$
8. return $\emptyset$

Correctness follows by the Substructure Lemma and induction.

Running time:
- Depth $k$, branching factor 2 $\Rightarrow$ total number of calls is $O(2^k)$
- $O(E)$ work per recursive call
- Total runtime: $O(2^k \cdot E)$.

Exponential in $k$, but much better than $\Theta(n^k)$ (i.e., still polynomial for $k = O(\log n)$)

IV. Covering Problems

Vertex Cover
Outline

Introduction

Vertex Cover

The Set-Covering Problem
The Set-Covering Problem

- Given: set \( X \) of size \( n \) and family of subsets \( \mathcal{F} \)
- Goal: Find a minimum-size subset \( C \subseteq \mathcal{F} \)

s.t. \( X = \bigcup_{S \in C} S \).

Only solvable if \( \bigcup_{S \in \mathcal{F}} S = X \)!

Remarks:
- generalisation of the vertex-cover problem and hence also NP-hard.
- models resource allocation problems
Greedy

**Strategy:** Pick the set \( S \) that covers the largest number of uncovered elements.

**Greedy-Set-Cover** \((X, \mathcal{F})\)

1. \( U = X \)
2. \( \mathcal{C} = \emptyset \)
3. while \( U \neq \emptyset \)
4. select an \( S \in \mathcal{F} \) that maximizes \( |S \cap U| \)
5. \( U = U - S \)
6. \( \mathcal{C} = \mathcal{C} \cup \{S\} \)
7. return \( \mathcal{C} \)

In the example of Figure 35.3, **Greedy-Set-Cover** adds to \( \mathcal{C} \), in order, the sets \( S_1, S_4, S_5 \), and then \( S_3 \) or \( S_6 \). The algorithm works as follows. The set \( U \) contains, at each stage, the set of remaining uncovered elements. The set \( \mathcal{C} \) contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset \( S \) that covers as many uncovered elements as possible (breaking ties arbitrarily). After \( S \) is selected, line 5 removes its elements from \( U \), and line 6 places \( S \) into \( \mathcal{C} \). When the algorithm terminates, the set \( \mathcal{C} \) contains a subfamily of \( \mathcal{F} \) that covers \( X \).

Exercise 35.3-3 asks for a linear-time algorithm.

**Analysis**

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the \( d \)th harmonic number \( H_d = \sum_{i=1}^{d} \frac{1}{i} \) (see Section A.1) by \( H.d/ \).

**Theorem 35.4**

**Greedy-Set-Cover** is a polynomial-time \( \frac{1}{\ln n} \)-approximation algorithm, where

\[
\text{opt} = H.\max f|S| \mid S \in \mathcal{F}g
\]

**Proof**

We have already shown that **Greedy-Set-Cover** runs in polynomial time.

Greedy chooses \( S_1, S_4, S_5 \) and \( S_3 \) (or \( S_6 \)), which is a cover of size 4.
Greedy Strategy: Pick the set $S$ that covers the largest number of uncovered elements.

**Greedy-Set-Cover** $(X, F)$

1. $U = X$
2. $C = \emptyset$
3. While $U \neq \emptyset$
   4. Select an $S \in F$ that maximizes $|S \cap U|$
   5. $U = U - S$
   6. $C = C \cup \{S\}$
4. Return $C$

Can be easily implemented to run in time polynomial in $|X|$ and $|F|$.

Optimal cover is $C = \{S_3, S_4, S_5\}$

How good is the approximation ratio?
Theorem 35.4

**GREEDY-SET-COVER** is a polynomial-time \( \rho(n) \)-algorithm, where

\[
\rho(n) = H(\max\{|S| : |S| \in \mathcal{F}\}) \leq \ln(n) + 1.
\]

\[
H(k) := \sum_{i=1}^{k} \frac{1}{k} \leq \ln(k) + 1
\]

**Idea:** Distribute cost of 1 for each added set over the newly covered elements.

**Definition of cost**

If an element \( x \) is covered for the first time by set \( S_i \) in iteration \( i \), then

\[
c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.
\]
Illustration of Costs

\[
\begin{align*}
\frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{6} + \frac{1}{3} + \frac{1}{3} + \frac{1}{2} + \frac{1}{2} + 1 &= 4
\end{align*}
\]
Proof of Theorem 35.4 (1/2)

Definition of cost

If \( x \) is covered for the first time by a set \( S_i \), then \( c_x := \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|} \).

Proof.

- Each step of the algorithm assigns one unit of cost, so

\[
|C| = \sum_{x \in X} c_x \tag{1}
\]

- Each element \( x \in X \) is in at least one set in the optimal cover \( C^* \), so

\[
\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x \tag{2}
\]

- Combining 1 and 2 gives

\[
|C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) \tag{\square}
\]

Key Inequality: \( \sum_{x \in S} c_x \leq H(|S|) \).
Proof of Theorem 35.4 (2/2)

Proof of the Key Inequality

\[ \sum_{x \in S} c_x \leq H(|S|) \]

Remaining uncovered elements in \( S \)  
Sets chosen by the algorithm

- For any \( S \in \mathcal{F} \) and \( i = 1, 2, \ldots, |\mathcal{C}| = k \) let \( u_i := |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i)| \)
  \[ u_0 \geq u_1 \geq \cdots \geq u_{|\mathcal{C}|} = 0 \]
  \( u_{i-1} - u_i \) counts the items covered first time by \( S_i \).

\[ \sum_{x \in S} c_x = \sum_{i=1}^k (u_{i-1} - u_i) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} \]

Each factor is at most one.

- Further, by definition of the \textsc{Greedy-Set-Cover}:
  \[ |S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}. \]

- Combining the last inequalities gives:

\[ \sum_{x \in S} c_x \leq \sum_{i=1}^k (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} = \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \]

\[ \leq \sum_{i=1}^k \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \]

\[ = \sum_{i=1}^k (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) = H(|S|). \]

IV. Covering Problems

The Set-Covering Problem
The same approach also gives an approximation ratio of $O(\ln(n))$ if there exists a cost function $c : S \to \mathbb{Z}^+$. 

**Theorem 35.4**

**Greedy-Set-Cover** is a polynomial-time $\rho(n)$-algorithm, where

$$\rho(n) = H(\max\{|S| : |S| \in F\}) \leq \ln(n) + 1.$$  

- Is the bound on the approximation ratio tight?
- Is there a better algorithm?

**Lower Bound**

Unless P=NP, there is no $c \cdot \ln(n)$ approximation algorithm for set cover for some constant $0 < c < 1$. 

IV. Covering Problems  The Set-Covering Problem
Example where Greedy is a \((1/2) \cdot \log_2 n\) factor off

**Instance**

- Given any integer \(k \geq 3\)
- There are \(n = 2^{k+1} - 2\) elements overall
- Sets \(S_1, S_2, \ldots, S_k\) are pairwise disjoint and each set contains \(2, 4, \ldots, 2^k\) elements
- Sets \(T_1, T_2\) are disjoint and each set contains half of the elements of each set \(S_1, S_2, \ldots, S_k\)

\[k = 4:\]

\[
\begin{array}{cccc}
S_1 & S_2 & S_3 & S_4 \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

\(T_1\) and \(T_2\)
Example where Greedy is a \((1/2) \cdot \log_2 n\) factor off

**Instance**
- Given any integer \(k \geq 3\)
- There are \(n = 2^{k+1} - 2\) elements overall
- Sets \(S_1, S_2, \ldots, S_k\) are pairwise disjoint and each set contains \(2, 4, \ldots, 2^k\) elements
- Sets \(T_1, T_2\) are disjoint and each set contains half of the elements of each set \(S_1, S_2, \ldots, S_k\)

\[ k = 4: \]

**Solution of Greedy consists of \(k\) sets.**
Example where Greedy is a \((1/2) \cdot \log_2 n\) factor off

- Given any integer \(k \geq 3\)
- There are \(n = 2^{k+1} - 2\) elements overall
- Sets \(S_1, S_2, \ldots, S_k\) are pairwise disjoint and each set contains 
  \(2, 4, \ldots, 2^k\) elements
- Sets \(T_1, T_2\) are disjoint and each set contains half of the elements of
  each set \(S_1, S_2, \ldots, S_k\)

\[ k = 4: \]

\(S_1\)
\(S_2\)
\(S_3\)
\(S_4\)

\(T_1\)
\(T_2\)

Optimum consists of 2 sets.
V. Approximation Algorithms via Exact Algorithms

Thomas Sauerwald
Outline

The Subset-Sum Problem

Parallel Machine Scheduling
The Subset-Sum Problem

- **Given:** Set of positive integers $S = \{x_1, x_2, \ldots, x_n\}$ and positive integer $t$
- **Goal:** Find a subset $S' \subseteq S$ which maximizes $\sum_{i: x_i \in S'} x_i \leq t$.

This problem is NP-hard

$t = 13$ tons

$\begin{align*} x_1 &= 10 \\
    x_2 &= 4 \\
    x_3 &= 5 \\
    x_4 &= 6 \\
    x_5 &= 1 \end{align*}$

$x_1 + x_5 = 11$
Given: Set of positive integers \( S = \{ x_1, x_2, \ldots, x_n \} \) and positive integer \( t \)

Goal: Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i: x_i \in S'} x_i \leq t \).

This problem is **NP-hard**

\[
\begin{align*}
x_1 &= 10 \\
x_2 &= 4 \\
x_3 &= 5 \\
x_4 &= 6 \\
x_5 &= 1 \\
\end{align*}
\]

\[
x_3 + x_4 + x_5 = 12
\]

\( t = 13 \text{ tons} \)
An Exact (Exponential-Time) Algorithm

Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum($S, t$)**
1. $n = |S|$
2. $L_0 = \langle 0 \rangle$
3. for $i = 1$ to $n$
   4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
   5. remove from $L_i$ every element that is greater than $t$
6. return the largest element in $L_n$

Example:
- $S = \{1, 4, 5\}$
- $L_0 = \langle 0 \rangle$
- $L_1 = \langle 0, 1 \rangle$
- $L_2 = \langle 0, 1, 4, 5 \rangle$
- $L_3 = \langle 0, 1, 4, 5, 6, 9, 10 \rangle$
Dynamic Programming: Compute bottom-up all possible sums $\leq t$

**Exact-Subset-Sum ($S$, $t$)**

1. $n = |S|
2. $L_0 = \langle 0 \rangle$
3. **for** $i = 1$ **to** $n$
4. $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$
5. remove from $L_i$ every element that is greater than $t$
6. **return** the largest element in $L_n$

- **Correctness:** $L_n$ contains all sums of $\{x_1, x_2, \ldots, x_n\}$
- **Runtime:** $O(2^1 + 2^2 + \cdots + 2^n) = O(2^n)$

There are $2^i$ subsets of $\{x_1, x_2, \ldots, x_i\}$. Better runtime if $t$ and/or $|L_i|$ are small.

can be shown by induction on $n$
Towards a FPTAS

Idea: Don’t need to maintain two values in \( L \) which are close to each other.

Trimming a List

- Given a trimming parameter \( 0 < \delta < 1 \)
- Trimming \( L \) yields minimal sublist \( L' \) so that for every \( y \in L \): \( \exists z \in L' : \)

\[
\frac{y}{1 + \delta} \leq z \leq y.
\]

\[\text{TRIM}(L, \delta)\]
1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. for \( i = 2 \) to \( m \)
5. if \( y_i > \text{last} \cdot (1 + \delta) \) \( \text{// } y_i \geq \text{last} \text{ because } L \text{ is sorted} \)
6. append \( y_i \) onto the end of \( L' \)
7. \( \text{last} = y_i \)
8. return \( L' \)

Trims list in time \( \Theta(m) \), if \( L \) is given in sorted order.

- \( L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \)
- \( \delta = 0.1 \)
- \( L' = \langle 10, 12, 15, 20, 23, 29 \rangle \)
Illustration of the Trim Operation

\textbf{TRIM}(L, \delta)
1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. \textbf{for} \( i = 2 \) \textbf{to} \( m \)
5. \hspace{1cm} \textbf{if} \( y_i > \text{last} \cdot (1 + \delta) \) \hspace{1cm} \( \parallel \) \( y_i \geq \text{last} \) because \( L \) is sorted
6. \hspace{1cm} \text{append} \( y_i \) onto the end of \( L' \)
7. \hspace{1cm} \text{last} = y_i
8. \textbf{return} \( L' \)

\( \delta = 0.1 \)

After the initialization (lines 1-3)

\( L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \)

\( L' = \langle 10 \rangle \)
Illustration of the Trim Operation

\text{TRIM}(L, \delta)

1. let \( m \) be the length of \( L \)
2. \( L' = \langle y_1 \rangle \)
3. \( \text{last} = y_1 \)
4. \textbf{for} \( i = 2 \) to \( m \)
5. \hspace{1em} \textbf{if} \( y_i > \text{last} \cdot (1 + \delta) \) \hspace{1em} // \( y_i \geq \text{last} \) because \( L \) is sorted
6. \hspace{2em} append \( y_i \) onto the end of \( L' \)
7. \hspace{2em} \text{last} = y_i
8. \textbf{return} \( L' \)

\[ \delta = 0.1 \]

The returned list \( L' \)

\[ L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \]

\[ L' = \langle 10, 12, 15, 20, 23, 29 \rangle \]
The FPTAS

**APPROX-SUBSET-SUM** \((S, t, \epsilon)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. **for** \(i = 1\) **to** \(n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. \(L_i = \text{TRIM}(L_i, \epsilon/2n)\)
6. remove from \(L_i\) every element that is greater than \(t\)
7. let \(z^*\) be the largest value in \(L_n\)
8. **return** \(z^*\)

**EXACT-SUBSET-SUM** \((S, t)\)

1. \(n = |S|\)
2. \(L_0 = \langle 0 \rangle\)
3. **for** \(i = 1\) **to** \(n\)
4. \(L_i = \text{MERGE-LISTS}(L_{i-1}, L_{i-1} + x_i)\)
5. remove from \(L_i\) every element that is greater than \(t\)
6. **return** the largest element in \(L_n\)

Repeated application of **TRIM** to make sure \(L_i\)'s remain short.

- We must bound the inaccuracy introduced by repeated trimming
- We must show that the algorithm is polynomial time
Running through an Example

Algorithm: \textsc{Approx-Subset-Sum}($S, t, \epsilon$)

\begin{enumerate}
\item $n = |S|$ \hfill \\
\item $L_0 = \langle 0 \rangle$ \hfill \\
\item \textbf{for} $i = 1$ \textbf{to} $n$ \hfill \\
\item \hspace{1em} $L_i = \text{Merge-Lists}(L_{i-1}, L_{i-1} + x_i)$ \hfill \\
\item \hspace{1em} $L_i = \text{Trim}(L_i, \epsilon/2n)$ \hfill \\
\item \hspace{1em} remove from $L_i$ every element that is greater than $t$ \hfill \\
\item \hspace{1em} let $z^*$ be the largest value in $L_n$ \hfill \\
\item \hspace{1em} \textbf{return} $z^*$ \hfill \\
\end{enumerate}

- **Input:** $S = \langle 104, 102, 201, 101 \rangle$, $t = 308$, $\epsilon = 0.4$

$\Rightarrow$ **Trimming parameter:** $\delta = \epsilon/(2 \cdot n) = \epsilon/8 = 0.05$

- line 2: $L_0 = \langle 0 \rangle$
- line 4: $L_1 = \langle 0, 104 \rangle$
- line 5: $L_1 = \langle 0, 104 \rangle$
- line 6: $L_1 = \langle 0, 104 \rangle$
- line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
- line 5: $L_2 = \langle 0, 102, 206 \rangle$
- line 6: $L_2 = \langle 0, 102, 206 \rangle$
- line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
- line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$
- line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$
- line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
- line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$
- line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$

Returned solution $z^* = 302$, which is 2% within the optimum $307 = 104 + 102 + 101$
Analysis of \textsc{Approx-Subset-Sum}

**Theorem 35.8**

\textsc{Approx-Subset-Sum} is a FPTAS for the subset-sum problem.

**Proof (Approximation Ratio):**

- Returned solution \( z^* \) is a valid solution ✓
- Let \( y^* \) denote an optimal solution
- For every possible sum \( y \leq t \) of \( x_1, \ldots, x_i \), there exists an element \( z \in L_i \) s.t.:

\[
\frac{y}{(1 + \epsilon/(2n))^i} \leq z \leq y \quad \implies \quad \frac{y^*}{(1 + \epsilon/(2n))^i} \leq z \leq y^*
\]

Can be shown by induction on \( i \)

and now using the fact that \( (1 + \frac{\epsilon/2}{n})^n \to e^\frac{\epsilon}{2} \) yields

\[
\frac{y^*}{z} \leq e^{\frac{\epsilon}{2}} \quad \text{Taylor approximation of } e
\]

\[
\leq 1 + \frac{\epsilon}{2} + (\frac{\epsilon}{2})^2 \leq 1 + \epsilon
\]
Analysis of \textsc{Approx-Subset-Sum}

\textbf{Theorem 35.8}

\textsc{Approx-Subset-Sum} is a FPTAS for the subset-sum problem.

\textbf{Proof (Running Time)}:

- \textbf{Strategy}: Derive a bound on $|L_i|$ (running time is polynomial in $|L_i|$)
- After trimming, two successive elements $z$ and $z'$ satisfy $z'/z \geq 1 + \epsilon/(2n)$

\Rightarrow Possible Values after trimming are 0, 1, and up to $\lfloor \log_{1+\epsilon/(2n)} t \rfloor$ additional values. Hence,

$$\log_{1+\epsilon/(2n)} t + 2 = \frac{\ln t}{\ln(1 + \epsilon/(2n))} + 2 \leq \frac{2n(1 + \epsilon/(2n)) \ln t}{\epsilon} + 2$$

For $x > -1$, $\ln(1 + x) \geq \frac{x}{1+x}$

\begin{align*}
\text{Need } \log(t) \text{ bits to represent } t \text{ and } n \text{ bits to represent } S.
\end{align*}

- This bound on $|L_i|$ is polynomial in the size of the input and in $1/\epsilon$. \qed
Concluding Remarks

The Subset-Sum Problem

- **Given:** Set of positive integers \( S = \{x_1, x_2, \ldots, x_n\} \) and positive integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which maximizes \( \sum_{i \in S'} x_i \leq t \).

Theorem 35.8

**APPROX-SUBSET-SUM** is a FPTAS for the subset-sum problem.

The Knapsack Problem

- **Given:** Items \( i = 1, 2, \ldots, n \) with weights \( w_i \) and values \( v_i \), and integer \( t \)
- **Goal:** Find a subset \( S' \subseteq S \) which
  1. maximizes \( \sum_{i \in S'} v_i \)
  2. satisfies \( \sum_{i \in S'} w_i \leq t \)

**Algorithm very similar to APPROX-SUBSET-SUM.**

There is a FPTAS for the Knapsack problem.

V. Approximation via Exact Algorithms  The Subset-Sum Problem  10
Outline

The Subset-Sum Problem

Parallel Machine Scheduling
Parallel Machine Scheduling

Machine Scheduling Problem

- Given: \( n \) jobs \( J_1, J_2, \ldots, J_n \) with processing times \( p_1, p_2, \ldots, p_n \), and \( m \) identical machines \( M_1, M_2, \ldots, M_m \)
- Goal: Schedule the jobs on the machines minimizing the makespan \( C_{\text{max}} = \max_{1 \leq j \leq n} C_j \), where \( C_k \) is the completion time of job \( J_k \).

- \( J_1: p_1 = 2 \)
- \( J_2: p_2 = 12 \)
- \( J_3: p_3 = 6 \)
- \( J_4: p_4 = 4 \)
Parallel Machine Scheduling

Machine Scheduling Problem

- **Given:** $n$ jobs $J_1, J_2, \ldots, J_n$ with processing times $p_1, p_2, \ldots, p_n$, and $m$ identical machines $M_1, M_2, \ldots, M_m$

- **Goal:** Schedule the jobs on the machines minimizing the makespan $C_{\text{max}} = \max_{1 \leq j \leq n} C_j$, where $C_k$ is the completion time of job $J_k$.

- $J_1$: $p_1 = 2$
- $J_2$: $p_2 = 12$
- $J_3$: $p_3 = 6$
- $J_4$: $p_4 = 4$

For the analysis, it will be convenient to denote by $C_i$ the completion time of a machine $i$. 

![Diagram showing the scheduling of jobs on machines](image_url)
Parallel Machine Scheduling is NP-complete even if there are only two machines.

**Proof Idea:** Polynomial time reduction from NUMBER-PARTITIONING.

Equivalent to the following **Online Algorithm** [CLRS]:
Whenever a machine is idle, schedule any job that has not yet been scheduled.

**LIST SCHEDULING** \((J_1, J_2, \ldots, J_n, m)\)

1. **while** there exists an unassigned job
2. Schedule job on the machine with the least load

How good is this most basic Greedy Approach?
List Scheduling Analysis (Observations)

Ex 35-5 a.&b.

a. The optimal makespan is at least as large as the greatest processing time, that is,

\[ C^*_\text{max} \geq \max_{1 \leq k \leq n} p_k. \]

b. The optimal makespan is at least as large as the average machine load, that is,

\[ C^*_\text{max} \geq \frac{1}{m} \sum_{k=1}^{n} p_k. \]

Proof:

a. The total processing times of all \( n \) jobs equals \( \sum_{k=1}^{n} p_k \)
b. One machine must have a load of at least \( \frac{1}{m} \cdot \sum_{k=1}^{n} p_k \)
List Scheduling Analysis (Final Step)

Ex 35-5 d. (Graham 1966)

For the schedule returned by the greedy algorithm it holds that

\[ C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k. \]

Hence list scheduling is a poly-time 2-approximation algorithm.

Proof:
- Let \( J_i \) be the last job scheduled on machine \( M_j \) with \( C_{\text{max}} = C_j \)
- When \( J_i \) was scheduled to machine \( M_j \), \( C_j - p_i \leq C_k \) for all \( 1 \leq k \leq m \)
- Averaging over \( k \) yields:

\[ C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{m} C_k = \frac{1}{m} \sum_{k=1}^{n} p_k \implies C_{\text{max}} \leq \frac{1}{m} \sum_{k=1}^{n} p_k + \max_{1 \leq k \leq n} p_k \leq 2 \cdot C_{\text{max}}^* \]

Using Ex 35-5 a. & b.
Improving Greedy Analysis can be shown to be almost tight. Is there a better algorithm?

The problem of the List-Scheduling Approach were the large jobs

Analysis can be shown to be almost tight. Is there a better algorithm?

**Least Processing Time** \( (J_1, J_2, \ldots, J_n, m) \)

1. Sort jobs decreasingly in their processing times
2. for \( i = 1 \) to \( m \)
3. \( C_i = 0 \)
4. \( S_i = \emptyset \)
5. end for
6. for \( j = 1 \) to \( n \)
7. \( i = \arg\min_{1 \leq k \leq m} C_k \)
8. \( S_i = S_i \cup \{j\}, C_i = C_i + p_j \)
9. end for
10. return \( S_1, \ldots, S_m \)

**Runtime:**
- \( O(n \log n) \) for sorting
- \( O(n \log m) \) for extracting the minimum (use priority queue).
Analysis of Improved Greedy

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

This can be shown to be tight (see next slide).

Proof (of approximation ratio 3/2).

- Observation 1: If there are at most $m$ jobs, then the solution is optimal.
- Observation 2: If there are more than $m$ jobs, then $C^*_\text{max} \geq 2 \cdot p_{m+1}$.
- As in the analysis for list scheduling, we have

$$C_j = (C_j - p_i) + p_i \leq C^*_\text{max} + \frac{1}{2} C^*_\text{max} = \frac{3}{2} C^*_\text{max}.$$ 

This is for the case $i \geq m + 1$ (otherwise, an even stronger inequality holds).
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines
- $n = 2m + 1$ jobs of length $2m - 1, 2m - 2, \ldots, m$ and one job of length $m$

$m = 5, n = 11$:

\[ C^*_{\text{max}} = 15 \]

\[ C_{\text{max}} = 19 \]
Tightness of the Bound for LPT

Graham 1966

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Proof of an instance which shows tightness:

- $m$ machines
- $n = 2m + 1$ jobs of length $2m - 1, 2m - 2, \ldots, m$ and one job of length $m$

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$
**Tightness of the Bound for LPT**

The LPT algorithm has an approximation ratio of $4/3 - 1/(3m)$.

Graham 1966

The tightness of this bound is demonstrated with the following example:

- **$m$** machines
- **$n = 2m + 1$** jobs of length $2m-1, 2m-2, \ldots, m$ and one job of length $m$

Proof of an instance which shows tightness:

$m = 5, n = 11$:

LPT gives $C_{\text{max}} = 19$

Optimum is $C^*_{\text{max}} = 15$

<table>
<thead>
<tr>
<th>$M_5$</th>
<th>5</th>
<th>5</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_4$</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>$M_3$</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>$M_2$</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>$M_1$</td>
<td>9</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

$C^*_{\text{max}} = 15$
A PTAS for Parallel Machine Scheduling

Basic Idea: For \((1 + \epsilon)-\)approximation, don’t have to work with exact \(p_k\)’s.

**SUBROUTINE** \((J_1, J_2, \ldots, J_n, m, T)\)

1. Either: **Return** a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\)
2. Or: **Return** there is no solution with makespan < \(T\)

**Key Lemma**

We will prove this on the next slides.

**SUBROUTINE** can be implemented in time \(n^{O(1/\epsilon^2)}\).

**Theorem (Hochbaum, Shmoys’87)**

There exists a **PTAS** for Parallel Machine Scheduling which runs in time \(O(n^{O(1/\epsilon^2)} \cdot \log P)\), where \(P := \sum_{k=1}^{n} p_k\).

**Proof (using Key Lemma):**

**PTAS** \((J_1, J_2, \ldots, J_n, m)\)

1. Do binary search to find smallest \(T\) s.t. \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_{\text{max}}\}\).
2. **Return** solution computed by **SUBROUTINE** \((J_1, J_2, \ldots, J_n, m, T)\)

Since \(0 \leq C^*_\text{max} \leq P\) and \(C^*_\text{max}\) is integral, binary search terminates after \(O(\log P)\) steps.
Implementation of Subroutine

**SUBROUTINE**(\(J_1, J_2, \ldots, J_n, m, T\))

1. Either: \textbf{Return} a solution with \(C_{\text{max}} \leq (1 + \epsilon) \cdot \max\{T, C^*_\text{max}\}\)
2. Or: \textbf{Return} there is no solution with makespan < \(T\)

---

**Observation**

Divide jobs into two groups: \(J_{\text{small}} = \{J_i: p_i \leq \epsilon \cdot T\}\) and \(J_{\text{large}} = J \setminus J_{\text{small}}\).

Given a solution for \(J_{\text{large}}\) only with makespan \((1 + \epsilon) \cdot T\), then greedily placing \(J_{\text{small}}\) yields a solution with makespan \((1 + \epsilon) \cdot \max\{T, C^*_\text{max}\}\).

---

**Proof:**

- Let \(M_j\) be the machine with largest load
- If there are no jobs from \(J_{\text{small}}\), then makespan is at most \((1 + \epsilon) \cdot T\).
- Otherwise, let \(i \in J_{\text{small}}\) be the last job added to \(M_j\).

\[
C_j - p_i \leq \frac{1}{m} \sum_{k=1}^{n} p_k \quad \Rightarrow \quad C_j \leq p_i + \frac{1}{m} \sum_{k=1}^{n} p_k \\
\leq \epsilon \cdot T + C^*_\text{max} \\
\leq (1 + \epsilon) \cdot \max\{T, C^*_\text{max}\} \quad \square
\]
Proof of Key Lemma

Use Dynamic Programming to schedule $J_{\text{large}}$ with makespan $(1 + \epsilon) \cdot T$.

- Let $b$ be the smallest integer with $1/b \leq \epsilon$. Define processing times $p'_i = \lceil p_j b^2 \rceil \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$.

  $\Rightarrow$ Every $p'_i = \alpha \cdot \frac{T}{b^2}$ for $\alpha = b, b + 1, \ldots, b^2$. Can assume there are no jobs with $p_i \geq T$!

- Let $C$ be all $(s_b, s_{b+1}, \ldots, s_{b^2})$ with $\sum_{i=j}^{b^2} s_i \cdot j \cdot \frac{T}{b^2} \leq T$.

  Assignments to one machine with makespan $\leq T$.

- Let $f(n_b, n_{b+1}, \ldots, n_{b^2})$ be the minimum number of machines required to schedule all jobs with makespan $\leq T$:

  $f(0, 0, \ldots, 0) = 0$

  $f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2})$.

  Assign some jobs to one machine, and then use as few machines as possible for the rest.

\[ \begin{align*}
1.5 \cdot T & \quad \text{\(\epsilon = 0.5\)} \\
1.25 \cdot T & \\
1.0 \cdot T & \\
0.75 \cdot T & \\
0.5 \cdot T & \\
0.25 \cdot T & \\
0 & \text{\(J_{\text{large}}\)} \\
\end{align*} \]

\[ \begin{align*}
1.5 \cdot T & \quad \text{\(b = 2\)} \\
1.25 \cdot T & \\
1.0 \cdot T & \\
0.75 \cdot T & \\
0.5 \cdot T & \\
0.25 \cdot T & \\
0 & \text{\(J_{\text{small}}\)} \\
\end{align*} \]

\[ \begin{align*}
1.5 \cdot T & \quad \text{\(p'_1\)} \\
1.25 \cdot T & \quad \text{\(p'_2\)} \\
1.0 \cdot T & \quad \text{\(p'_3\)} \\
0.75 \cdot T & \\
0.5 \cdot T & \\
0.25 \cdot T & \\
0 & \text{\(J_{\text{large}}\)} \\
\end{align*} \]
Proof of Key Lemma

Use Dynamic Programming to schedule \( J_{\text{large}} \) with makespan \((1 + \epsilon) \cdot T\).

- Let \( b \) be the smallest integer with \( 1/b \leq \epsilon \). Define processing times \( p'_i = \left\lfloor \frac{p_j b^2}{T} \right\rfloor \cdot \frac{T}{b^2} \).

- Every \( p'_i = \alpha \cdot \frac{T}{b^2} \) for \( \alpha = b, b + 1, \ldots, b^2 \).

- Let \( C \) be all \((s_b, s_{b+1}, \ldots, s_{b^2})\) with \( \sum_{i=j}^{b^2} s_j \cdot j \cdot \frac{T}{b^2} \leq T \).

- Let \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \) be the minimum number of machines required to schedule all jobs with makespan \( \leq T \):

\[
f(0, 0, \ldots, 0) = 0 \]
\[
f(n_b, n_{b+1}, \ldots, n_{b^2}) = 1 + \min_{(s_b, s_{b+1}, \ldots, s_{b^2}) \in C} f(n_b - s_b, n_{b+1} - s_{b+1}, \ldots, n_{b^2} - s_{b^2}).
\]

- Number of table entries is at most \( n^{b^2} \), hence filling all entries takes \( n^{O(b^2)} \).

- If \( f(n_b, n_{b+1}, \ldots, n_{b^2}) \leq m \) (for the jobs with \( p' \)), then return yes, otherwise no.

- As every machine is assigned at most \( b \) jobs \((p'_i \geq \frac{T}{b})\) and the makespan is \( \leq T \),

\[
C_{\text{max}} \leq T + b \cdot \max_{i \in J_{\text{large}}} (p_i - p'_i)
\]
\[
\leq T + b \cdot \frac{T}{b^2} \leq (1 + \epsilon) \cdot T.
\]

\(\square\)
**Final Remarks**

**Graham 1966**

List scheduling has an approximation ratio of 2.

**Graham 1966**

The LPT algorithm has an approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

**Theorem (Hochbaum, Shmoys'87)**

There exists a PTAS for Parallel Machine Scheduling which runs in time $O(n^{O(1/\epsilon^2)} \cdot \log P)$, where $P := \sum_{k=1}^{n} p_k$.

---

Can we find a FPTAS (for polynomially bounded processing times)? **No!**

Because for sufficiently small approximation ratio $1 + \epsilon$, the computed solution has to be optimal.
VI. Approximation Algorithms: Travelling Salesman Problem

Thomas Sauerwald
Outline

Introduction

General TSP

Metric TSP
The Traveling Salesman Problem (TSP)

Given a set of cities along with the cost of travel between them, find the cheapest route visiting all cities and returning to your starting point.

Formal Definition

- **Given**: A complete undirected graph \( G = (V, E) \) with nonnegative integer cost \( c(u, v) \) for each edge \((u, v) \in E\)
- **Goal**: Find a hamiltonian cycle of \( G \) with minimum cost.

Solution space consists of \( n! \) possible tours!

Actually the right number is \((n - 1)!/2\)

Special Instances

- **Metric TSP**: costs satisfy triangle inequality:

\[
\forall u, v, w \in V : c(u, w) \leq c(u, v) + c(v, w).
\]

- **Euclidean TSP**: cities are points in the Euclidean space, costs are equal to their Euclidean distance

Even this version is NP hard (Ex. 35.2-2)
History of the TSP problem (1954)

Dantzig, Fulkerson and Johnson found an optimal tour through 42 cities.

http://www.math.uwaterloo.ca/tsp/history/img/dantzig_big.html
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)
The Dantzig-Fulkerson-Johnson Method

1. Create a linear program (variable $x(u, v) = 1$ iff tour goes between $u$ and $v$)
2. Solve the linear program. If the solution is integral and forms a tour, stop. Otherwise find a new constraint to add (cutting plane)

More cuts are needed to find integral solution
Outline

Introduction

General TSP

Metric TSP
Hardness of Approximation

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.

**Theorem 35.3**

Proof: **Idea:** Reduction from the hamiltonian-cycle problem.

- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \( (u, v) \in E' \):

\[
  c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise.}
  \end{cases}
\]

Can create representations of \( G' \) and \( c \) in time polynomial in \(|V|\) and \(|E|!\)

Large weight will render this edge useless!

\( G = (V, E) \) \hspace{1cm} \text{Reduction} \hspace{1cm} \rho \cdot 4 + 1 \hspace{1cm} G' = (V, E') \)
Hardness of Approximation

**Theorem 35.3**

If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho$ for the general TSP.

**Proof:** *Idea: Reduction from the hamiltonian-cycle problem.*

- Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem.
- Let $G' = (V, E')$ be a complete graph with costs for each $(u, v) \in E'$:
  
  $$c(u, v) = \begin{cases} 
  1 & \text{if } (u, v) \in E, \\
  \rho |V| + 1 & \text{otherwise}.
  \end{cases}$$

- If $G$ has a hamiltonian cycle $H$, then $(G', c)$ contains a tour of cost $|V|$.

\[G = (V, E)\]

\[G' = (V, E')\]
Hardness of Approximation

Theorem 35.3

If \( P \neq NP \), then for any constant \( \rho \geq 1 \), there is no polynomial-time approximation algorithm with approximation ratio \( \rho \) for the general TSP.


- Let \( G = (V, E) \) be an instance of the hamiltonian-cycle problem.
- Let \( G' = (V, E') \) be a complete graph with costs for each \( (u, v) \in E' \):

\[
c(u, v) = \begin{cases} 
1 & \text{if } (u, v) \in E, \\
\rho |V| + 1 & \text{otherwise.}
\end{cases}
\]

- If \( G \) has a hamiltonian cycle \( H \), then \( (G', c) \) contains a tour of cost \( |V| \).
- If \( G \) does not have a hamiltonian cycle, then any tour \( T \) must use some edge \( \notin E \),

\[
\Rightarrow c(T) \geq (\rho |V| + 1) + (|V| - 1) = (\rho + 1)|V|.
\]

- Gap of \( \rho + 1 \) between tours which are using only edges in \( G \) and those which don’t.
- \( \rho \)-Approximation of TSP in \( G' \) computes hamiltonian cycle in \( G \) (if one exists).
Proof of Theorem 35.3 from a higher perspective

General Method to prove inapproximability results!

All instances with a hamiltonian cycle

All instances with cost $\leq k$

All instances with cost $\geq \rho \cdot k$

instances of Hamilton

instances of TSP

VI. Travelling Salesman Problem
Outline

Introduction

General TSP

Metric TSP
The TSP Problem with the Triangle Inequality

Idea: First compute an MST, and then create a tour based on the tree.

Algorithm: APPROX-TSP-TOUR \((G, c)\)

1. Select a vertex \(r \in G.V\) to be a "root" vertex
2. Compute a minimum spanning tree \(T\) for \(G\) from root \(r\)
   using MST-PRIM\((G, c, r)\)
3. Let \(H\) be a list of vertices, ordered according to when they are first visited
   in a preorder tree walk of \(T\)
4. Return the Hamiltonian cycle \(H\)

Runtime is dominated by MST-PRIM, which is \(\Theta(V^2)\).
Run of APPROX-TSP-TOUR

1. Compute MST

Solution has cost $\approx 19.704$ - not optimal!

Better solution, yet still not optimal!

This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST ✓
2. Perform preorder walk on MST

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

1. Compute MST ✓
2. Perform preorder walk on MST ✓
3. Return list of vertices according to the preorder tree walk

Solution has cost $\approx 19.704$ - not optimal!
Better solution, yet still not optimal!
This is the optimal solution (cost $\approx 14.715$).
Run of APPROX-TSP-TOUR

Solution has cost $\approx 19.704$ - not optimal!

1. Compute MST ✓
2. Perform preorder walk on MST ✓
3. Return list of vertices according to the preorder tree walk ✓
Run of \textsc{Approx-Tsp-Tour}

Better solution, yet \textit{still} not optimal!

1. Compute MST ✓
2. Perform preorder walk on MST ✓
3. Return list of vertices according to the preorder tree walk ✓
Run of APPROX-TSP-TOUR

This is the optimal solution (cost $\approx 14.715$).

1. Compute MST ✓
2. Perform preorder walk on MST ✓
3. Return list of vertices according to the preorder tree walk ✓
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:
- Consider the optimal tour $H^*$ and remove one edge.
- This yields a spanning tree and therefore $c(T) \leq c(H^*)$.
- Exploiting that all edge costs are non-negative.

Exploiting the triangle inequality.

Solution $H$ of APPROX-TSP

Spanning tree as a subset of $H^*$
Proof of the Approximation Ratio

Theorem 35.2

APPROX-TSP-TOUR is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

Proof:
- Consider the optimal tour $H^*$ and remove one edge
  $\Rightarrow$ yields a spanning tree and therefore $c(T) \leq c(H^*)$
- Let $W$ be the full walk of the spanning tree $T$ (including repeated visits)
  $\Rightarrow$ Full walk traverses every edge exactly twice, so
  $$c(W) = 2c(T) \leq 2c(H^*)$$

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

Optimal solution $H^*$
Proof of the Approximation Ratio

**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time 2-approximation for the traveling-salesman problem with the triangle inequality.

**Proof:**

- Consider the optimal tour $H^*$ and remove one edge
  ⇒ yields a spanning tree and therefore $c(T) \leq c(H^*)$
- Let $W$ be the full walk of the spanning tree $T$ (including repeated visits)
  ⇒ Full walk traverses every edge exactly twice, so
    \[ c(W) = 2c(T) \leq 2c(H^*) \]
  - Deleting duplicate vertices from $W$ yields a tour $H$ with smaller cost:
    \[ c(H) \leq c(W) \leq 2c(H^*) \]

Walk $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$

optimal solution $H^*$

Exploiting triangle inequality!
**Theorem 35.2**

**APPROX-TSP-TOUR** is a polynomial-time **2-approximation** for the traveling-salesman problem with the triangle inequality.

---

**CHRISTOFIDES**\((G, c)\)
1: select a vertex \(r \in G \cdot V\) to be a “root” vertex
2: compute a minimum spanning tree \(T\) for \(G\) from root \(r\)
3: \hspace{1cm} using MST-PRIM\((G, c, r)\)
4: compute a perfect matching \(M\) with minimum weight in the complete graph
5: \hspace{1cm} over the odd-degree vertices in \(T\)
6: let \(H\) be a list of vertices, ordered according to when they are first visited
7: \hspace{1cm} in a Eulearian circuit of \(T \cup M\)
8: **return** \(H\)

---

**Theorem (Christofides’76)**

There is a polynomial-time \(\frac{3}{2}\)-approximation algorithm for the travelling salesman problem with the triangle inequality.
Run of Christofides

1. Compute MST

Solution has cost $\approx 15.54$ within 10% of the optimum!
Run of CHRISTOFIDES

1. Compute MST ✓
2. Add a minimum-weight perfect matching $M$ of the odd vertices in $T$ ✓
Run of Christofides

Solution has cost $\approx 15.54$ - within 10% of the optimum!

1. Compute MST ✓
2. Add a minimum-weight perfect matching $M$ of the odd vertices in $T$ ✓
3. Find an Eulerian Circuit ✓
Run of CHRISTOFIDES

Solution has cost \( \approx 15.54 \) - within 10% of the optimum!

1. Compute MST \( \checkmark \)
2. Add a minimum-weight perfect matching \( M \) of the odd vertices in \( T \) \( \checkmark \)
3. Find an Eulerian Circuit \( \checkmark \)
4. Transform the Circuit into a Hamiltonian Cycle \( \checkmark \)
Concluding Remarks

Theorem (Christofides’76)
There is a polynomial-time $\frac{3}{2}$-approximation algorithm for the travelling salesman problem with the triangle inequality.

Both received the Gödel Award 2010

Theorem (Arora’96, Mitchell’96)
There is a PTAS for the Euclidean TSP Problem.

“Christos Papadimitriou told me that the traveling salesman problem is not a problem. It’s an addiction.”

Jon Bentley 1991
VII. Approximation Algorithms: Randomisation and Rounding

Thomas Sauerwald
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover
Performance Ratios for Randomised Approximation Algorithms

Approximation Ratio

A randomised algorithm for a problem has approximation ratio $\rho(n)$, if for any input of size $n$, the expected cost $C$ of the returned solution and optimal cost $C^*$ satisfy:

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n).$$

Call such an algorithm randomised $\rho(n)$-approximation algorithm.

Approximation Schemes

An approximation scheme is an approximation algorithm, which given any input and $\epsilon > 0$, is a $(1 + \epsilon)$-approximation algorithm.

- It is a polynomial-time approximation scheme (PTAS) if for any fixed $\epsilon > 0$, the runtime is polynomial in $n$. For example, $O(n^2/\epsilon)$.
- It is a fully polynomial-time approximation scheme (FPTAS) if the runtime is polynomial in both $1/\epsilon$ and $n$. For example, $O((1/\epsilon)^2 \cdot n^3)$.
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover
MAX-3-CNF Satisfiability

- **Given:** 3-CNF formula, e.g.: \((x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor \overline{x_3} \lor x_5) \land \cdots\)
- **Goal:** Find an assignment of the variables that satisfies as many clauses as possible.

Relaxation of the satisfiability problem. Want to compute how “close” the formula to being satisfiable is.

Example:

\((x_1 \lor x_3 \lor \overline{x_4}) \land (x_1 \lor \overline{x_3} \lor x_5) \land (x_2 \lor \overline{x_4} \lor x_5) \land (\overline{x_1} \lor x_2 \lor \overline{x_3})\)

\(x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0\) and \(x_5 = 1\) satisfies 3 (out of 4 clauses)

**Idea:** What about assigning each variable independently at random?
Analysis

Theorem 35.6

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a randomised \( 8/7 \)-approximation algorithm.

Proof:

- For every clause \( i = 1, 2, \ldots, m \), define a random variable:
  \[
  Y_i = 1 \{ \text{clause } i \text{ is satisfied} \}
  \]

- Since each literal (including its negation) appears at most once in clause \( i \),
  \[
  \Pr \left[ \text{clause } i \text{ is not satisfied} \right] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}
  \]

  \( \Rightarrow \quad \Pr \left[ \text{clause } i \text{ is satisfied} \right] = 1 - \frac{1}{8} = \frac{7}{8} \]

  \[
  \Rightarrow \quad \mathbb{E} [Y_i] = \Pr [Y_i = 1] \cdot 1 = \frac{7}{8}.
  \]

- Let \( Y := \sum_{i=1}^{m} Y_i \) be the number of satisfied clauses. Then,
  \[
  \mathbb{E} [Y] = \mathbb{E} \left[ \sum_{i=1}^{m} Y_i \right] = \sum_{i=1}^{m} \mathbb{E} [Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7}{8} \cdot m.
  \]

  \( \square \)
Interesting Implications

**Theorem 35.6**
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( \frac{8}{7} \)-approximation algorithm.

**Corollary**
For any instance of MAX-3-CNF, there exists an assignment which satisfies at least \( \frac{7}{8} \) of all clauses.

There is \( \omega \in \Omega \) such that \( Y(\omega) \geq \mathbb{E}[Y] \! \).

Probabilistic Method: powerful tool to show existence of a non-obvious property.

**Corollary**
Any instance of MAX-3-CNF with at most 7 clauses is satisfiable.

Follows from the previous Corollary.
Expected Approximation Ratio

**Theorem 35.6**
Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a polynomial-time randomised \( 8/7 \)-approximation algorithm.

One could prove that the probability to satisfy \( (7/8) \cdot m \) clauses is at least \( 1/(8m) \).

\[
E[ Y ] = \frac{1}{2} \cdot E[ Y \mid x_1 = 1 ] + \frac{1}{2} \cdot E[ Y \mid x_1 = 0 ].
\]

\( Y \) is defined as in the previous proof.

One of the two conditional expectations is greater than \( E[ Y ] \)!

**GREEDY-3-CNF(\( \phi, n, m \))**
1: for \( j = 1, 2, \ldots, n \)
2: Compute \( E[ Y \mid x_1 = v_1 \ldots, x_{j-1} = v_{j-1}, x_j = 1 ] \)
3: Compute \( E[ Y \mid x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 0 ] \)
4: Let \( x_j = v_j \) so that the conditional expectation is maximized
5: return the assignment \( v_1, v_2, \ldots, v_n \)
Analysis of \textbf{GREEDY-3-CNF}(\(\phi, n, m\))

\textbf{Theorem}

\textbf{GREEDY-3-CNF}(\(\phi, n, m\)) is a polynomial-time \(8/7\)-approximation.

\textbf{Proof:}

- \textbf{Step 1:} polynomial-time algorithm
  - In iteration \(j = 1, 2, \ldots, n\), \(Y = Y(\phi)\) averages over \(2^{n-j+1}\) assignments
  - A smarter way is to use linearity of (conditional) expectations:

  \[ \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1] = \sum_{i=1}^{m} \mathbb{E}[Y_i | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = 1] \]

- \textbf{Step 2:} satisfies at least \(7/8 \cdot m\) clauses
  - Due to the greedy choice in each iteration \(j = 1, 2, \ldots, n\),
  \[
  \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}, x_j = v] \geq \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-1} = v_{j-1}] \\
  \geq \mathbb{E}[Y | x_1 = v_1, \ldots, x_{j-2} = v_{j-2}] \\
  \vdots \\
  \geq \mathbb{E}[Y] = \frac{7}{8} \cdot m.
  \]

This algorithm is deterministic.
Run of **GREEDY-3-CNF**($\varphi, n, m$)

\[(\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor \overline{x_4}) \land (x_1 \lor x_2 \lor \overline{x_4}) \land (\overline{x_1} \lor x_3 \lor x_4) \land (x_1 \lor x_2 \lor \overline{x_4}) \land (\overline{x_1} \lor x_2 \lor \overline{x_3}) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x_3} \lor \overline{x_4})\]

VII. Randomisation and Rounding MAX-3-CNF
Run of \textsc{Greedy-3-CNF}(\(\varphi, n, m\))

\[ 1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land (\overline{x_2} \lor \overline{x_3}) \land (x_2 \lor x_3) \land (\overline{x_2} \lor x_3) \land 1 \land (x_2 \lor \overline{x_3} \lor \overline{x_4}) \]

\[ \begin{align*}
&\text{Root: } 8.75 \\
&\text{Branch A: } x_1 = 0 \\
&\text{Branch B: } x_1 = 1 \\
&\text{Branch C: } 8.625 \\
&\text{Branch D: } 8.875 \\
&\text{Branch E: } 8.75 \\
&\text{Branch F: } 9 \\
&\text{Branch G: } 8.75 \\
&\end{align*} \]
Run of \textbf{GREEDY-3-CNF}(\(\varphi, n, m\))

\[ 1 \land 1 \land 1 \land (\overline{x_3} \lor x_4) \land 1 \land 1 \land (x_3) \land 1 \land 1 \land (\overline{x_3} \lor \overline{x_4}) \]

\[ x_1 = 0 \quad \text{8.625} \quad x_1 = 1 \]

\[ x_2 = 0 \quad \text{8.75} \quad x_2 = 1 \]

\[ x_3 = 0 \quad x_3 = 1 \]

\[ x_4 = 0 \quad x_4 = 1 \]

\[ 000? \quad 001? \quad 010? \quad 011? \quad 100? \quad 101? \quad 110? \quad 111? \]

\[ 000 \quad 001 \quad 001 \quad 001 \quad 010 \quad 010 \quad 011 \quad 011 \]

\[ 010 \quad 011 \quad 011 \quad 011 \quad 100 \quad 101 \quad 110 \quad 111 \]

\[ 100 \quad 101 \quad 110 \quad 111 \]

\[ 110 \quad 111 \quad 111 \quad 111 \]

\[ \text{Return of solution satisfies 9 out of 10 clauses, but the formula is satisfiable.} \]
Run of GREEDY-3-CNF(\( \varphi, n, m \))

\[ 1 \land 1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1 \]

VII. Randomisation and Rounding MAX-3-CNF
Run of **GREEDY-3-CNF** ($\varphi, n, m$)

$1 \land 1 \land 1 \land 1 \land 1 \land 0 \land 1 \land 1 \land 1$

VII. Randomisation and Rounding

MAX-3-CNF
Run of \textbf{GREEDY-3-CNF}(\(\varphi, n, m\))

\[(x_1 \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor x_4) \land (x_1 \lor x_2 \lor \overline{x_4}) \land (x_1 \lor \overline{x_3} \lor \overline{x_4}) \land (x_1 \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_3} \lor x_4) \land (\overline{x_1} \lor x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor x_3) \land (x_1 \lor x_3 \lor x_4) \land (x_2 \lor \overline{x_3} \lor \overline{x_4})\]

\[
\begin{array}{c}
\text{Returned solution satisfies 9 out of 10 clauses, but the formula is satisfiable.}
\end{array}
\]
**Theorem 35.6**

Given an instance of MAX-3-CNF with \( n \) variables \( x_1, x_2, \ldots, x_n \) and \( m \) clauses, the randomised algorithm that sets each variable independently at random is a *randomised 8/7-approximation algorithm*.

**Theorem**

\[ \text{GREEDY-3-CNF}(\phi, n, m) \text{ is a polynomial-time 8/7-approximation.} \]

**Theorem (Hastad’97)**

For any \( \epsilon > 0 \), there is no polynomial time \( 8/7 - \epsilon \) approximation algorithm of MAX3-SAT unless P=NP.

Roughly speaking, there is nothing smarter than just guessing.
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover
The **Weighted Vertex-Cover Problem**

**Vertex Cover Problem**

- **Given:** Undirected, *vertex-weighted* graph $G = (V, E)$
- **Goal:** Find a *minimum-weight* subset $V' \subseteq V$ such that if $(u, v) \in E(G)$, then $u \in V'$ or $v \in V'$.

This is (still) an NP-hard problem.

**Applications:**

- Every *edge* forms a *task*, and every *vertex* represents a *person/machine* which can execute that task
- **Weight** of a vertex could be *salary* of a person
- Perform all tasks with the *minimal amount of resources*
The Greedy Approach from (Unweighted) Vertex Cover

**APPROX-VERTEX-COVER** *(G)*

1. \( C = \emptyset \)
2. \( E' = G \cdot E \)
3. \( \text{while } E' \neq \emptyset \)
   4. let \((u, v)\) be an arbitrary edge of \( E' \)
   5. \( C = C \cup \{u, v\} \)
   6. remove from \( E' \) every edge incident on either \( u \) or \( v \)
7. return \( C \)

---

**Figure 35.1**

The operation of **APPROX-VERTEX-COVER**.

(a) The input graph \( G \), which has 7 vertices and 8 edges.
(b) The edge \( \{b, c\} \), shown heavy, is the first edge chosen by **APPROX-VERTEX-COVER**. Vertices \( b \) and \( c \), shown lightly shaded, are added to the set \( C \) containing the vertex cover being created. Edges \( \{a, b\} \), \( \{c, e\} \), and \( \{c, d\} \), shown as dashed, are removed since they are now covered by some vertex in \( C \).
(c) Edge \( \{e, f\} \) is chosen; vertices \( e \) and \( f \) are added to \( C \).
(d) Edge \( \{d, g\} \) is chosen; vertices \( d \) and \( g \) are added to \( C \).
(e) The set \( C \), which is the vertex cover produced by **APPROX-VERTEX-COVER**, contains the vertices \( b, c, d, e, f, g \).
(f) The optimal vertex cover for this problem contains only three vertices: \( b, d, \) and \( e \).

**Computed solution has weight 101.**
The Greedy Approach from (Unweighted) Vertex Cover

**APPROX-VERTEX-COVER** $(G)$

1. $C = \emptyset$
2. $E' = G.E$
3. while $E' \neq \emptyset$
4. let $(u, v)$ be an arbitrary edge of $E'$
5. $C = C \cup \{u, v\}$
6. remove from $E'$ every edge incident on either $u$ or $v$
7. return $C$

**Computed solution** has weight 100.

**Optimal solution** has weight 4.
Invoking an (Integer) Linear Program

Idea: Round the solution of an associated linear program.

0-1 Integer Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v)x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in \{0, 1\} \quad \text{for each } v \in V
\end{align*}
\]

Optimum is a lower bound on the optimal weight of a minimum weight-cover.

Rounding Rule: if \(x(v) \geq 1/2\) then round up, otherwise round down.

Linear Program

\[
\begin{align*}
\text{minimize} & \quad \sum_{v \in V} w(v)x(v) \\
\text{subject to} & \quad x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \\
& \quad x(v) \in [0, 1] \quad \text{for each } v \in V
\end{align*}
\]
The Algorithm

**Approx-Min-Weight-VC** \((G, w)\)

1. \( C = \emptyset \)
2. compute \( \bar{x} \), an optimal solution to the linear program
3. for each \( v \in V \)
   4. if \( \bar{x}(v) \geq 1/2 \)
   5. \( C = C \cup \{v\} \)
6. return \( C \)

**Theorem 35.7**

**Approx-Min-Weight-VC** is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.
Example of \textsc{Approx-Min-Weight-VC}

\[
\bar{x}(a) = \bar{x}(b) = \bar{x}(e) = \frac{1}{2}, \quad \bar{x}(d) = 1, \quad \bar{x}(c) = 0
\]

\[
x(a) = x(b) = x(e) = 1, \quad x(d) = 1, \quad x(c) = 0
\]

fractional solution of LP with weight $= 5.5$

rounded solution of LP with weight $= 10$

optimal solution with weight $= 6$
Approximation Ratio

Proof (Approximation Ratio is 2):

- Let \( C^* \) be an optimal solution to the minimum-weight vertex cover problem.
- Let \( z^* \) be the value of an optimal solution to the linear program, so
  \[
  z^* \leq w(C^*)
  \]

- **Step 1:** The computed set \( C \) covers all vertices:
  - Consider any edge \((u, v) \in E\) which imposes the constraint \( x(u) + x(v) \geq 1 \)
  \[\Rightarrow\] at least one of \( \overline{x}(u) \) and \( \overline{x}(v) \) is at least 1/2 \[\Rightarrow\] \( C \) covers edge \((u, v)\)

- **Step 2:** The computed set \( C \) satisfies \( w(C) \leq 2z^* \):
  \[
  w(C^*) \geq z^* = \sum_{v \in V} w(v)\overline{x}(v) \geq \sum_{v \in V: \overline{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \frac{1}{2}w(C). \]

VII. Randomisation and Rounding

Weighted Vertex Cover 18
Outline

Randomised Approximation

MAX-3-CNF

Weighted Vertex Cover

Weighted Set Cover
The **Weighted Set-Covering Problem**

- **Given:** set \( X \) and a family of subsets \( \mathcal{F} \), and a cost function \( c : \mathcal{F} \to \mathbb{R}^+ \)
- **Goal:** Find a minimum-cost subset \( C \subseteq \mathcal{F} \)

\[
\text{s.t.} \quad X = \bigcup_{S \in C} S.
\]

**Remarks:**
- generalisation of the weighted vertex-cover problem
- models resource allocation problems
### Setting up an Integer Program

**0-1 Integer Program**

\[
\begin{align*}
\text{minimize} & \quad \sum_{S \in \mathcal{F}} c(S)y(S) \\
\text{subject to} & \quad \sum_{S \in \mathcal{F}: x \in S} y(S) \geq 1 \quad \text{for each } x \in X \\
& \quad y(S) \in \{0, 1\} \quad \text{for each } S \in \mathcal{F}
\end{align*}
\]

### Linear Program

**Linear Program**

\[
\begin{align*}
\text{minimize} & \quad \sum_{S \in \mathcal{F}} c(S)y(S) \\
\text{subject to} & \quad \sum_{S \in \mathcal{F}: x \in S} y(S) \geq 1 \quad \text{for each } x \in X \\
& \quad y(S) \in [0, 1] \quad \text{for each } S \in \mathcal{F}
\end{align*}
\]
Back to the Example

The strategy employed for Vertex-Cover would take all 6 sets!

Even worse: If all \( y \)'s were below \( 1/2 \), we would not even return a valid cover!

Cost equals 8.5

<table>
<thead>
<tr>
<th></th>
<th>( S_1 )</th>
<th>( S_2 )</th>
<th>( S_3 )</th>
<th>( S_4 )</th>
<th>( S_5 )</th>
<th>( S_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( y(\cdot) )</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>
Randomised Rounding

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$y(.)$</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Idea: Interpret the $y$-values as probabilities for picking the respective set.

Lemma

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies

\[
\mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)
\]

- The probability that an element $x \in X$ is covered satisfies

\[
\Pr \left[ x \in \bigcup_{S \in C} S \right] \geq 1 - \frac{1}{e}.
\]
Proof of Lemma

Let \( C \subseteq \mathcal{F} \) be a random subset with each set \( S \) being included independently with probability \( y(S) \).

- The expected cost satisfies \( E[ c(C) ] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \).
- The probability that \( x \) is covered satisfies \( \Pr[ x \in \bigcup_{S \in C} S ] \geq 1 - \frac{1}{e} \).

Proof:

- **Step 1**: The expected cost of the random set \( C 
\[
E[ c(C) ] = E \left[ \sum_{S \in C} c(S) \right] = \sum_{S \in \mathcal{F}} \Pr[ S \in C ] \cdot c(S) = \sum_{S \in \mathcal{F}} y(S) \cdot c(S).
\]

- **Step 2**: The probability for an element to be (not) covered
\[
\Pr[ x \not\in \bigcup_{S \in C} S ] = \prod_{S \in \mathcal{F} : x \in S} \Pr[ S \not\in C ] = \prod_{S \in \mathcal{F} : x \in S} (1 - y(S)) \leq \prod_{S \in \mathcal{F} : x \in S} e^{-y(S)} = e^{-\sum_{S \in \mathcal{F} : x \in S} y(S)} \leq e^{-1}
\]

1 + \( x \) \( \leq e^{x} \) for any \( x \in \mathbb{R} \)
The Final Step

**Lemma**

Let $C \subseteq \mathcal{F}$ be a random subset with each set $S$ being included independently with probability $y(S)$.

- The expected cost satisfies $\mathbb{E}[c(C)] = \sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
- The probability that $x$ is covered satisfies $\Pr[x \in \bigcup_{S \in C} S] \geq 1 - \frac{1}{e}$.

**Problem:** Need to make sure that every element is covered!

**Idea:** Amplify this probability by taking the union of $\Omega(\log n)$ random sets $C$.

**Weighted Set Cover-LP** ($X$, $\mathcal{F}$, $c$)

1. compute $y$, an optimal solution to the linear program
2. $C = \emptyset$
3. repeat $2 \ln n$ times
4. for each $S \in \mathcal{F}$
5. let $C = C \cup \{S\}$ with probability $y(S)$
6. return $C$

Clearly runs in polynomial-time!
Analysis of Weighted Set Cover-LP

Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

Proof:

- **Step 1:** The probability that $C$ is a cover
  - By previous Lemma, an element $x \in X$ is covered in one of the $2 \ln n$ iterations with probability at least $1 - \frac{1}{e}$, so that
    \[
    \Pr \left[ x \notin \bigcup_{S \in C} S \right] \leq \left( \frac{1}{e} \right)^{2 \ln n} = \frac{1}{n^2}.
    \]
  - This implies for the event that all elements are covered:
    \[
    \Pr \left[ X = \bigcup_{S \in C} S \right] = 1 - \Pr \left[ \bigcup_{x \in X} \{ x \notin \bigcup_{S \in C} S \} \right] \geq 1 - n \cdot \frac{1}{n^2} = 1 - \frac{1}{n}.
    \]

- **Step 2:** The expected approximation ratio
  - By previous lemma, the expected cost of one iteration is $\sum_{S \in \mathcal{F}} c(S) \cdot y(S)$.
  - Linearity $\Rightarrow E \left[ c(C) \right] \leq 2 \ln(n) \cdot \sum_{S \in \mathcal{F}} c(S) \cdot y(S) \leq 2 \ln(n) \cdot c(C^*)$.
Theorem

- With probability at least $1 - \frac{1}{n}$, the returned set $C$ is a valid cover of $X$.
- The expected approximation ratio is $2 \ln(n)$.

By Markov's inequality, $\Pr[c(C) \leq 4 \ln(n) \cdot c(C^*)] \geq 1/2$.

Hence with probability at least $1 - \frac{1}{n} - \frac{1}{2} > \frac{1}{3}$, solution is within a factor of $4 \ln(n)$ of the optimum.

Typical Approach for Designing Approximation Algorithms based on LPs

Probability could be further increased by repeating
VIII. Approximation Algorithms: MAX-CUT Problem

Thomas Sauerwald
Simple Algorithms for MAX-CUT

A Solution based on Semidefinite Programming

Summary
Max-Cut

Given: Undirected graph $G = (V, E)$
Goal: Find a subset $S \subseteq V$ such that $|E(S, V \setminus S)|$ is maximized.

### Weighted MAX-CUT

- Every edge $e \in E$ has a non-negative weight $w(e)$.

### MAX-CUT Problem

- **Given:** Undirected graph $G = (V, E)$
- **Goal:** Find a subset $S \subseteq V$ such that $|E(S, V \setminus S)|$ is maximized.

### Weighted MAX-CUT

Maximize the weights of edges crossing the cut, i.e., maximize $w(S) := \sum_{\{u, v\} \in E(S, V \setminus S)} w(\{u, v\})$

**Applications:**
- cluster analysis
- VLSI design

---

$S = \{a, b, g\}$
$w(S) = 18$
Suppose that for each vertex $v$, we randomly and independently place $v$ in $S$ with probability $1/2$ and in $V \setminus S$ with probability $1/2$. Then this algorithm is a randomized 2-approximation algorithm.

We could employ the same derandomisation used for MAX-3-CNF.

Proof: We express the expected weight of the random cut $(S, V \setminus S)$ as:

$$E[w(S, V \setminus S)]$$

$$= \mathbb{E} \left[ \sum_{\{u,v\} \in E(S, V \setminus S)} w(\{u, v\}) \right]$$

$$= \sum_{\{u,v\} \in E} \Pr[\{u \in S \cap v \in (V \setminus S)\} \cup \{u \in (V \setminus S) \cap v \in S\}] \cdot w(\{u, v\})$$

$$= \sum_{\{u,v\} \in E} \left( \frac{1}{4} + \frac{1}{4} \right) \cdot w(\{u, v\})$$

$$= \frac{1}{2} \sum_{\{u,v\} \in E} w(\{u, v\}) \geq \frac{1}{2} w^*.$$ 

VIII. MAX-CUT Problem

Simple Algorithms for MAX-CUT
Local Search

Local Search: Switch side of a vertex if it increases the cut.

Local Search \( (G, w) \)
1: Let \( S \) be an arbitrary subset of \( V \)
2: do
3: \( \text{flag} = 0 \)
4: if \( \exists u \in S \) with \( w(S \setminus \{u\}, (V \setminus S) \cup \{u\}) \geq w(S, V \setminus S) \) then
5: \( S = S \setminus \{u\} \)
6: \( \text{flag} = 1 \)
7: end if
8: if \( \exists u \in V \setminus S \) with \( w(S \cup \{u\}, (V \setminus S) \setminus \{u\}) \geq w(S, V \setminus S) \) then
9: \( S = S \cup \{u\} \)
10: \( \text{flag} = 1 \)
11: end if
12: while \( \text{flag} = 1 \)
13: return \( S \)
Illustration of Local Search

Step 1: Move $a$ into $S$

Step 2: Move $g$ into $S$

Step 3: Move $d$ into $S$

Step 4: Move $b$ into $S$

Step 5: Move $a$ into $V / S$ (local search terminates)

After Step 5: Local Search terminates

A better solution could be found:

$Cut = 0$
$Cut = 5$
$Cut = 8$
$Cut = 10$
$Cut = 11$
$Cut = 12$
$Cut = 13$
Illustration of Local Search

Step 1: Move a into S

Cut = 5

Step 2: Move g into S

Step 3: Move d into S

Step 4: Move b into S

Step 5: Move a into V (local search terminates)

A better solution could be found:

Cut = 0
Cut = 0
Cut = 5
Cut = 8
Cut = 10
Cut = 10
Cut = 11
Cut = 11
Cut = 12

VIII. MAX-CUT Problem Simple Algorithms for MAX-CUT
Illustration of Local Search

Step 1: Move $a$ into $S$

Step 2: Move $g$ into $S$

Step 3: Move $d$ into $S$

Step 4: Move $b$ into $S$

Step 5: Move $a$ into $V$

(local search terminates)

After Step 5: Local Search terminates

A better solution could be found:

Cut = 0
Cut = 0
Cut = 5
Cut = 5
Cut = 8
Cut = 8
Cut = 10
Cut = 10
Cut = 11
Cut = 11
Cut = 12
Cut = 13

Step 2: Move $g$ into $S$

Cut = 8
Illustration of Local Search

Step 1: Move a into S
Step 2: Move g into S
Step 3: Move d into S
Step 4: Move b into S
Step 5: Move a into V

(local search terminates)

After Step 5: Local Search terminates

A better solution could be found:

Cut = 0
Cut = 0
Cut = 5
Cut = 5
Cut = 8
Cut = 8
Cut = 10
Cut = 10
Cut = 11
Cut = 11
Cut = 12
Cut = 13

Step 3: Move d into S
Cut = 10
Illustration of Local Search

Step 1: Move $a$ into $S$
Step 2: Move $g$ into $S$
Step 3: Move $d$ into $S$
Step 4: Move $b$ into $S$
Step 5: Move $a$ into $V$

(local search terminates)

After Step 5: Local Search terminates

A better solution could be found:

$\text{Cut} = 0$
$\text{Cut} = 0$
$\text{Cut} = 5$
$\text{Cut} = 5$
$\text{Cut} = 8$
$\text{Cut} = 8$
$\text{Cut} = 10$
$\text{Cut} = 11$
$\text{Cut} = 11$
$\text{Cut} = 12$

Step 4: Move $b$ into $S$

$\text{Cut} = 11$
Illustration of Local Search

Step 1: Move $a$ into $S$

Step 2: Move $g$ into $S$

Step 3: Move $d$ into $S$

Step 4: Move $b$ into $S$

Step 5: Move $a$ into $V \setminus S$

(local search terminates)

Cut $= 12$
Illustration of Local Search

Step 1: Move a into S
Step 2: Move g into S
Step 3: Move d into S
Step 4: Move b into S
Step 5: Move a into V

(local search terminates)

After Step 5: Local Search terminates

A better solution could be found:

Cut = 1
Cut = 0
Cut = 5
Cut = 8
Cut = 10
Cut = 11
Cut = 12
Cut = 13

A better solution could be found:

Cut = 13
Theorem

The cut returned by **LOCAL-SEARCH** satisfies \( W \geq (1/2)W^* \).

Proof:

- At the time of termination, for every vertex \( u \in S \):
  \[
  \sum_{v \in V \setminus S, v \sim u} w(\{u, v\}) \geq \sum_{v \in S, v \sim u} w(\{u, v\}),
  \]

- Similarly, for any vertex \( u \in V \setminus S \):
  \[
  \sum_{v \in S, v \sim u} w(\{u, v\}) \geq \sum_{v \in V \setminus S, v \sim u} w(\{u, v\}).
  \]

- Adding up equation 1 for all vertices in \( S \) and equation 2 for all vertices in \( V \setminus S \),
  \[
  w(S) \geq 2 \cdot \sum_{v \in S, u \in S, u \sim v} w(\{u, v\}) \quad \text{and} \quad w(S) \geq 2 \cdot \sum_{v \in V \setminus S, u \in V \setminus S, u \sim v} w(\{u, v\}).
  \]

- Adding up these two inequalities, and diving by 2 yields
  \[
  w(S) \geq \sum_{v \in S, u \in S, u \sim v} w(\{u, v\}) + \sum_{v \in V \setminus S, u \in V \setminus S, u \sim v} w(\{u, v\}).
  \]

Every edge appears on one of the two sides.
The cut returned by LOCAL-SEARCH satisfies $W \geq (1/2)W^*$. 

What is the running time of LOCAL-SEARCH?

- **Unweighted Graphs:** Cut increases by at least one in each iteration $\Rightarrow$ at most $n^2$ iterations
- **Weighted Graphs:** could take exponential time in $n$ (not obvious...)

VIII. MAX-CUT Problem Simple Algorithms for MAX-CUT
Outline

Simple Algorithms for MAX-CUT

A Solution based on Semidefinite Programming

Summary
Max-Cut Problem

**High-Level-Approach:**
1. Describe the Max-Cut Problem as a quadratic optimisation problem
2. Solve a corresponding semidefinite program that is a relaxation of the original problem
3. Recover an approximation for the original problem from the approximation for the semidefinite program

**Quadratic program**

\[
\begin{align*}
\text{maximize} & & \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - y_i y_j) \\
\text{subject to} & & y_i \in \{-1, +1\}, \quad i = 1, \ldots, n.
\end{align*}
\]

Label vertices by 1, 2, \ldots, \(n\) and express weight function etc. as a \(n \times n\)-matrix.

This models the MAX-CUT problem

\[
S = \{i \in V : y_i = +1\}, \quad V \setminus S = \{i \in V : y_i = -1\}
\]
Relaxation

**Quadratic program**

\[
\text{maximize} \quad \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - y_i y_j) \\
\text{subject to} \quad y_i \in \{-1, +1\}, \quad i = 1, \ldots, n.
\]

Any solution of the original program can be recovered by setting \(v_i = (y_i, 0, 0, \ldots, 0)\)!

**Vector Programming Relaxation**

\[
\text{maximize} \quad \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - v_i v_j) \\
\text{subject to} \quad v_i \cdot v_i = 1, \quad i = 1, \ldots, n. \\
v_i \in \mathbb{R}^n
\]
Positive Definite Matrices

Definition

A matrix $A \in \mathbb{R}^{n \times n}$ is positive semidefinite iff for all $y \in \mathbb{R}^n$,

$$y^T \cdot A \cdot y \geq 0.$$ 

Remark

1. $A$ is symmetric and positive definite iff there exists a $n \times n$ matrix $B$ with $B^T \cdot B = A$.
2. If $A$ is symmetric and positive definite, then the matrix $B$ above can be computed in polynomial time.

Examples:

Using Cholesky-decomposition

$$A = \begin{pmatrix} 18 & 2 \\ 2 & 6 \end{pmatrix} = \begin{pmatrix} 4 & -1 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 4 & 1 \\ -1 & 2 \end{pmatrix}, \quad \text{so } A \text{ is SPD.}$$

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \quad \text{since } (1 \ -1) \cdot \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} = -2, \quad A \text{ is not SPD.}$$
Reformulating the Quadratic Program as a Semidefinite Program

**Vector Programming Relaxation**

Maximize: \[ \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - v_i v_j) \]

Subject to:
\[ v_i \cdot v_i = 1 \]
\[ v_i \in \mathbb{R}^n \]

**Reformulation:**
- Introduce \( n^2 \) variables \( a_{i,j} = v_i \cdot v_j \), which give rise to a matrix \( A \)
- If \( V \) is the matrix given by the vectors \((v_1, v_2, \ldots, v_n)\), then \( A = V^T \cdot V \) is symmetric and positive definite

**Semidefinite Program**

Maximize: \[ \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - a_{i,j}) \]

Subject to:
\[ A = (a_{i,j}) \text{ is symmetric and positive definite,} \]
\[ a_{i,i} = 1 \text{ for all } i = 1, \ldots, n \]

Solve this (which can be done in polynomial time), and recover \( V \) using Cholesky Decomposition.
Rounding the Vector Program

Vector Programming Relaxation

maximize \[ \frac{1}{2} \sum_{(i,j) \in E} w_{i,j} \cdot (1 - v_i v_j) \]
subject to \[ v_i \cdot v_i = 1 \quad i = 1, \ldots, n. \]
\[ v_i \in \mathbb{R}^n \]

Rounding by a random hyperplane:
1. Pick a random vector \( r = (r_1, r_2, \ldots, r_n) \) by drawing each component from \( \mathcal{N}(0, 1) \)
2. Put \( i \in V \) if \( v_i \cdot r \geq 0 \) and \( i \in V \setminus S \) otherwise

The probability that two vectors \( v_i, v_j \in \mathbb{R}^n \) are separated by the (random) hyperplane given by \( r \) equals \( \frac{\arccos(v_i \cdot v_j)}{\pi} \).

Follows by projecting on the plane given by \( v_i \) and \( v_j \).
Illustration of the Hyperplane
A second (technical) Lemma

Lemma 2

For any $x \in [-1, 1]$,

$$
\frac{1}{\pi} \arccos(x) \geq 0.878 \cdot \frac{1}{2} (1 - x).
$$

![Graph of $f(x) = \frac{1}{\pi} \arccos(x)$ and $\frac{1}{2} (1 - x)$]
Theorem (Goemans, Williamson’96)

The algorithm has an approximation ratio of $\frac{1}{0.878} \approx 1.139$.

Proof: Define an indicator variable

$$X_{i,j} = \begin{cases} 
1 & \text{if } (i,j) \in E \text{ are on different sides of the hyperplane} \\
0 & \text{otherwise.}
\end{cases}$$

Hence for the (random) weight of the computed cut,

$$E[w(S)] = \sum_{\{i,j\} \in E} X_{i,j}$$

$$= \sum_{\{i,j\} \in E} E[X_{i,j}]$$

$$= \sum_{\{i,j\} \in E} w_{i,j} \cdot Pr[\{i,j\} \in E \text{ is in the cut}]$$

By Lemma 1

$$= \sum_{\{i,j\} \in E} w_{i,j} \cdot \frac{1}{\pi} \arccos(v_i \cdot v_j)$$

By Lemma 2

$$\geq 0.878 \cdot \frac{1}{2} \sum_{\{i,j\} \in E} w_{i,j} \cdot (1 - v_i \cdot v_j) = 0.878 \cdot z^* \geq 0.878 \cdot W^*.$$
There is a randomised polynomial-time 1.139-approximation algorithm for MAX-CUT. Similar approach can be applied to MAX-3-CNF and yields an approximation ratio of 1.345.

Unless P=NP, there is no $\rho$-approximation algorithm for MAX-CUT with $\rho \leq \frac{17}{16} = 1.0625$.

Assuming the so-called Unique Games Conjecture holds, unless P=NP there is no $\rho$-approximation algorithm for MAX-CUT with

$$\rho \leq \max_{-\frac{1}{\pi} \leq x \leq \frac{1}{\pi}} \left\{ \frac{1}{2} \left(1 - x\right) \right\} \leq 1.139$$
Other Approximation Algorithms for MAX-CUT

Theorem (Mathieu, Schudy’08)

For any $\epsilon > 0$, there is a randomised algorithm with running time $O(n^2)2^{O(1/\epsilon^2)}$ so that the expected value of the output deviates from the maximum cut value by at most $O(\epsilon \cdot n^2)$. This is an additive approximation!

Algorithm (1):

1. Take a sample $S$ of $x = O(1/\epsilon^2)$ vertices chosen uniformly at random
2. For each of the $2^x$ possible cuts, go through vertices in $V \setminus S$ in random order and place them on the side of the cut which maximizes the crossing edges
3. Output the best cut found

Theorem (Trevisan’08)

There is a randomised 1.833-approximation algorithm for MAX-CUT which runs in $O(n^2 \cdot \text{polylog}(n))$ time.

Exploits relation between the smallest eigenvalue and the structure of the graph.
Outline

Simple Algorithms for MAX-CUT

A Solution based on Semidefinite Programming

Summary
Spectrum of Approximations

- MAX-CLIQUE
- SET-COVER
- VERTEX-COVER, MAX-3-CNF, MAX-CUT, METRIC-TSP
- SCHEDULING, EUCLIDEAN-TSP
- KNAPSACK, SUBSET-SUM
- FPTAS, PTAS, APX, log-APX, poly-APX