Contents I

1. Colorings I
   - Cycle
   - Trees

2. Eulerian cycle
   - Introduction
   - The algorithm

3. Matchings
   - Introduction
   - Algorithm
   - Running times

4. Colorings II

5. MIS
   - Introduction
   - MIS and Coloring

6. Coloring III
   - Outerplanar graphs

7. Simulations
   - Simple simulations
   - Advanced simulations

- Bipartite graphs
- $\Delta + 1$ Coloring any Graph
Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$. 
Colorings

Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute [exists?] Function $c : V \mapsto \{1, \cdots, k\}$ with:
Colorings

**Coloring Problem**

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute [exists?] Function $c : V \mapsto \{1, \cdots, k\}$ with:
  
  $\forall \{a, b\} \in E : c(a) \neq c(b)$. 
Colorings

**Coloring Problem**

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute $\exists$ Function $c : V \mapsto \{1, \cdots, k\}$ with:
  - $\forall \{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  \[
  \chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall \{a, b\} \in E : c(a) \neq c(b)\}.
  \]
Colorings

Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute $\exists$ Function $c : V \mapsto \{1, \cdots, k\}$ with:
  - $\forall \{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  $$\chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall \{a, b\} \in E : c(a) \neq c(b)\}.$$
Colorings

Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute $\exists$ Function $c : V \mapsto \{1, \ldots, k\}$ with:
  - $\forall\{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  $\chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \ldots, k\} \mid \forall\{a, b\} \in E : c(a) \neq c(b)\}$.

- Coloring problem is NP-complete.
Colorings

**Coloring Problem**

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute [exists?] Function $c : V \mapsto \{1, \cdots, k\}$ with:
  - $\forall\{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  $\chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall\{a, b\} \in E : c(a) \neq c(b)\}$.

- Coloring problem is NP-complete.
- Let $G = C_n$, i.e. $G = (\{v_0, \cdots, v_{n-1}\}, \{v_i, v_{(i+1) \mod n}\} \mid 0 \leq i < n)$. 
Colorings

Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute [exists?] Function $c : V \mapsto \{1, \cdots, k\}$ with:
  - $\forall\{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  \[ \chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall\{a, b\} \in E : c(a) \neq c(b)\} \]

- Coloring problem is NP-complete.
- Let $G = C_n$, i.e. $G = (\{v_0, \cdots, v_{n-1}\}, \{v_i, v_{(i+1) \mod n}\} \mid 0 \leq i < n)$.
- Then we have $\chi(C_n) \leq 3$ and $\chi(C_{2\cdot n}) \leq 2$ ($\chi(C_{2\cdot n+1}) = 3$).
Given undirected graph \( G = (V, E) \) and \( k \in \mathbb{N} \).
Compute \([\text{exists?}]\) Function \( c : V \mapsto \{1, \cdots, k\} \) with:
\[ \forall\{a, b\} \in E : c(a) \neq c(b). \]
Coloring number (chromatic index) of \( G \):
\[ \chi(G) := \min\{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall\{a, b\} \in E : c(a) \neq c(b)\}. \]

- Coloring problem is NP-complete.
- Let \( G = C_n \), i.e. \( G = (\{v_0, \cdots, v_{n-1}\}, \{v_i, v_{(i+1) \text{ mod } n}\} \mid 0 \leq i < n) \).
- Then we have \( \chi(C_n) \leq 3 \) and \( \chi(C_{2\cdot n}) \leq 2 \) \( (\chi(C_{2\cdot n+1}) = 3) \).
- We do not have a nice order on the nodes:
Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.

Compute function $c : V \mapsto \{1, \cdots, k\}$ with:

- $\forall \{a, b\} \in E : c(a) \neq c(b)$.

Coloring number (chromatic index) of $G$:

$$\chi(G) := \min \{k \mid \exists c : V \mapsto \{1, \cdots, k\} \mid \forall \{a, b\} \in E : c(a) \neq c(b)\}.$$ 

Coloring problem is NP-complete.

Let $G = C_n$, i.e. $G = (\{v_0, \cdots, v_{n-1}\}, \{\{v_i, v_{(i+1) \mod n}\} \mid 0 \leq i < n\})$.

Then we have $\chi(C_n) \leq 3$ and $\chi(C_{2 \cdot n}) \leq 2$ ($\chi(C_{2 \cdot n+1}) = 3$).

We do not have a nice order on the nodes:

- let $\pi(i)$ be a permutation.
Colorings

Coloring Problem

- Given undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
- Compute [exists?] Function $c : V \mapsto \{1, \cdots, k\}$ with:
  - $\forall\{a, b\} \in E : c(a) \neq c(b)$.
- Coloring number (chromatic index) of $G$:
  $$\chi(G) := \min\{k | \exists c : V \mapsto \{1, \cdots, k\} | \forall\{a, b\} \in E : c(a) \neq c(b)\}.$$

- Coloring problem is NP-complete.
- Let $G = C_n$, i.e. $G = (\{v_0, \cdots, v_{n-1}\}, \{\{v_i, v_{(i+1) \mod n}\} | 0 \leq i < n\})$.
- Then we have $\chi(C_n) \leq 3$ and $\chi(C_{2\cdot n}) \leq 2$ ($\chi(C_{2\cdot n+1}) = 3$).
- We do not have a nice order on the nodes:
  - let $\pi(i)$ be a permutation
- Let $G = C_n$, i.e.
  $$G = (\{v_0, \cdots, v_{n-1}\}, \{\{v_{\pi(i)}, v_{\pi((i+1) \mod n)}\} | 0 \leq i < n\}).$$
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$. 
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$. 

Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$.
- Register $N_i$ holds $\pi(i)$. 
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i-1)$.
- Register $N_i$ holds $\pi(i)$.
- In register $C_i$ will be the color of $v_{R_i}$.
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i-1)$.
- Register $N_i$ holds $\pi(i)$.
- In register $C_i$ will be the color of $v_{R_i}$.
- Initialize $C_i$ with $i$. 
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i-1)$.
- Register $N_i$ holds $\pi(i)$.
- In register $C_i$ will be the color of $v_{R_i}$.
- Initialize $C_i$ with $i$.
- Reduce step by step the number of colors.
Parallel Coloring Algorithm of (on) a cycle (Idea)

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.  
- Register $R_i$ holds $\pi(i - 1)$.  
- Register $N_i$ holds $\pi(i)$.  
- In register $C_i$ will be the color of $v_{R_i}$.  
- Initialize $C_i$ with $i$.  
- Reduce step by step the number of colors.  
- We will use the colors $\{0, 1, \cdots, n\}$. 
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

\begin{verbatim}
for all \( P_{i+1} \) where \( 0 \leq i < n \) do in parallel
    \( \pi(i - 1) \rightarrow R_i \)
\end{verbatim}
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

\textbf{for all } P_{i+1} \textbf{ where } 0 \leq i < n \textbf{ do in parallel}
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i
\end{align*}
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle
for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$
$\pi(i) \rightarrow N_i$
$c = i$
$c \rightarrow C_i$
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i-1) \rightarrow R_i$

$\pi(i) \rightarrow N_i$

$c = i$

$c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

$C_{N_i} \rightarrow c'$

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.

$c = 2 \cdot k + ((c \gg k) \% 2)$.

$c \rightarrow C_i$
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$

$\pi(i) \rightarrow N_i$

$c = i$

$c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

$C_{N_i} \rightarrow c'$

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.

$c = 2 \cdot k + ((c \gg k) \% 2)$.

$c \rightarrow C_i$
Parallel Coloring Algorithm of (on) a cycle (Idea)

- At the start we are using $n$ colors.
Parallel Coloring Algorithm of (on) a cycle (Idea)

- At the start we are using \( n \) colors.
- Within each color-reduction will the coloring stay correct.
Parallel Coloring Algorithm of (on) a cycle (Idea)

- At the start we are using \( n \) colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from \( x \) to \( \log(x) + O(1) \).
Parallel Coloring Algorithm of (on) a cycle (Idea)

- At the start we are using $n$ colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from $x$ to $\log(x) + O(1)$.
- After $\lceil \log^*(n) \rceil$ reductions steps will be the coloring numbers $\leq 5$. 
Parallel Coloring Algorithm of (on) a cycle (Idea)

- At the start we are using $n$ colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from $x$ to $\log(x) + O(1)$.
- After $\lceil \log^*(n) \rceil$ reductions steps will be the coloring numbers $\leq 5$.
- A second reduction of colors will follow now:
**Last Steps**

- The rows hold $c$ and the columns hold $c'$.
- The entries in the table hold the new $c$.

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Last Steps

- The rows hold $c$ and the columns hold $c'$.

- The entries in the table hold the new $c$.

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
<td>010</td>
<td>000</td>
<td>100</td>
<td>000</td>
<td>010</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
<td>010</td>
<td>001</td>
<td>100</td>
<td>001</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>001</td>
<td>011</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>101</td>
<td>000</td>
<td>010</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>001</td>
<td>101</td>
<td>001</td>
<td>010</td>
<td>001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>011</td>
<td>000</td>
<td>101</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>001</td>
<td>011</td>
<td>001</td>
<td>101</td>
<td>001</td>
<td>011</td>
<td>001</td>
</tr>
</tbody>
</table>
Last Steps

- The rows hold $c$ and the columns hold $c'$.
- The entries in the table hold the new $c$.

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Last Steps

- The rows hold $c$ and the columns hold $c'$.
- The entries in the table hold the new $c$.

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- We only have the colors 000, 001, 010, 011, 100, 101 ($\leq 5$).
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

\[ \text{for all } P_{i+1} \text{ where } 0 \leq i < n \text{ do in parallel} \]

\[ \pi(i - 1) \rightarrow R_i \]
\[ \pi(i) \rightarrow N_i \]
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

\[
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i \\
c & = i \\
c & \rightarrow C_i
\end{align*}
\]
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

\textbf{for all} $P_{i+1}$ where $0 \leq i < n$ \textbf{do in parallel}

\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i \\
c & = i \\
c & \rightarrow C_i
\end{align*}

\textbf{repeat} $\lceil \log^*(n) \rceil + 2$ \textbf{times}

\begin{align*}
C_{N_i} & \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k) \% 2) & \neq ((c' \gg k) \% 2). \\
c & = 2 \cdot k + ((c \gg k) \% 2). \\
c & \rightarrow C_i
\end{align*}
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

\[
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i \\
c & = i \\
c & \rightarrow C_i
\end{align*}
\]

repeat $\lceil \log^*(n) \rceil + 2$ times

\[
\begin{align*}
C_{N_i} & \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k)\%2) & \neq ((c' \gg k)\%2).
\end{align*}
\]

\[
\begin{align*}
c & = 2 \cdot k + ((c \gg k)\%2).
\end{align*}
\]

\[
\begin{align*}
c & \rightarrow C_i
\end{align*}
\]

for $r := 5$ downto 3 do:
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

\[ \pi(i - 1) \rightarrow R_i \]
\[ \pi(i) \rightarrow N_i \]
\[ c = i \]
\[ c \rightarrow C_i \]

repeat $\lceil \log^*(n) \rceil + 2$ times

\[ C_{N_i} \rightarrow c' \]

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.
\[ c = 2 \cdot k + ((c \gg k) \% 2). \]
\[ c \rightarrow C_i \]

for $r := 5$ downto 3 do:

if $c = r$ then
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

\begin{align*}
\pi(i - 1) &\rightarrow R_i \\
\pi(i) &\rightarrow N_i \\
c &\equiv i \\
c &\rightarrow C_i
\end{align*}

repeat \lceil \log^*(n) \rceil + 2 times

\begin{align*}
C_{N_i} &\rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k) \% 2) \neq ((c' \gg k) \% 2). \\
c &\equiv 2 \cdot k + ((c \gg k) \% 2). \\
c &\rightarrow C_i
\end{align*}

for $r := 5$ downto 3 do:

if $c = r$ then

\begin{align*}
C_{N_i} &\rightarrow c' \\
c' &\rightarrow C_i \\
C_{N_i} &\rightarrow c''
\end{align*}
Parallel Coloring Algorithm of (on) a cycle (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

- $\pi(i - 1) \rightarrow R_i$
- $\pi(i) \rightarrow N_i$
- $c = i$
- $c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

- $C_{N_i} \rightarrow c'$
- minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.
- $c = 2 \cdot k + ((c \gg k) \% 2)$.
- $c \rightarrow C_i$

for $r := 5$ downto 3 do:

if $c = r$ then

- $C_{N_i} \rightarrow c'$
- $c' \rightarrow C_i$
- $C_{N_i} \rightarrow c''$
- $c := \min(\{0, 1, 2\} \setminus \{c', c''\})$
- $c \rightarrow C_i$
Coloring a Cycle

Theorem:
A cycle with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.
Coloring a Cycle

**Theorem:**
A cycle with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.

**Theorem:**
A cycle of $n$ processors may color itself in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.
Coloring a Cycle

**Theorem:**
A cycle with \( n \) nodes could be colored with \( n \) processors in time \( O(\log^* n) \) with at most 3 colors.

Proof: see above.

**Theorem:**
A cycle of \( n \) processors may color itself in time \( O(\log^* n) \) with at most 3 colors.

Proof: see above.

**Theorem:**
A cycle of \( n \) processors needs at least \( \Omega(\log^* n) \) time to color itself with at most 3 colors.

Proof: see V4.
Theorem:
A cycle with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.

Theorem:
A cycle of $n$ processors may color itself in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.

Theorem:
A cycle of $n$ processors needs at least $\Omega(\log^* n)$ time to color itself with at most 3 colors.

Proof: see V4.
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$. 
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$. 
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i-1)$.
- Register $N_i$ holds $\pi(j-1)$ where $j$ is the father of $i$. 
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$.
- Register $N_i$ holds $\pi(j - 1)$ where $j$ is the father of $i$.
- The father of the root $r$ is $r$. 
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i-1)$.
- Register $N_i$ holds $\pi(j-1)$ where $j$ is the father of $i$.
- The father of the root $r$ is $r$.
- In register $C_i$ will be the color of $v_{R_i}$.
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$.
- Register $N_i$ holds $\pi(j - 1)$ where $j$ is the father of $i$.
- The father of the root $r$ is $r$.
- In register $C_i$ will be the color of $v_{R_i}$.
- Initialize $C_i$ with $i$. 
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$.
- Register $N_i$ holds $\pi(j - 1)$ where $j$ is the father of $i$.
- The father of the root $r$ is $r$.
- In register $C_i$ will be the color of $v_{R_i}$.
- Initialize $C_i$ with $i$.
- Reduce step by step the number of colors.
Coloring a Tree

- A processor $P_i$ works on $v_{\pi(i-1)}$ for some permutation $\pi$.
- Register $R_i$ holds $\pi(i - 1)$.
- Register $N_i$ holds $\pi(j - 1)$ where $j$ is the father of $i$.
- The father of the root $r$ is $r$.
- In register $C_i$ will be the color of $v_{R_i}$.
- Initialize $C_i$ with $i$.
- Reduce step by step the number of colors.
- We will use the colors $\{0, 1, \cdots, n\}$.
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i-1) \rightarrow R_i$

$\pi(i) \rightarrow N_i$
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

\[
\text{for all } P_{i+1} \text{ where } 0 \leq i < n \text{ do in parallel}
\]
\[
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i \\
c & = i \\
c & \rightarrow C_i
\end{align*}
\]
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$
$\pi(i) \rightarrow N_i$
$c = i$
$c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

$C_{N_i} \rightarrow c'$

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.

$c = 2 \cdot k + ((c \gg k) \% 2)$.
$c \rightarrow C_i$
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$

$\pi(i) \rightarrow N_i$

$c = i$

$c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

$C_{N_i} \rightarrow c'$

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.

$c = 2 \cdot k + ((c \gg k) \% 2)$.

$c \rightarrow C_i$
Parallel Coloring Algorithm of (on) a tree (Idea)

- At the start we are using \( n \) colors.
Parallel Coloring Algorithm of (on) a tree (Idea)

- At the start we are using \( n \) colors.
- Within each color-reduction will the coloring stay correct.
Parallel Coloring Algorithm of (on) a tree (Idea)

- At the start we are using $n$ colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from $x$ to $\log(x) + O(1)$. 
Parallel Coloring Algorithm of (on) a tree (Idea)

- At the start we are using $n$ colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from $x$ to $\log(x) + O(1)$.
- After $\lceil \log^*(n) \rceil$ reductions steps will be the coloring numbers $\leq 5$. 
Parallel Coloring Algorithm of (on) a tree (Idea)

- At the start we are using $n$ colors.
- Within each color-reduction will the coloring stay correct.
- Within each color reduction will the coloring number be reduced from $x$ to $\log(x) + O(1)$.
- After $\lceil \log^*(n) \rceil$ reductions steps will be the coloring numbers $\leq 5$.
- A second reduction of colors will follow now:
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-tree

for all \( P_{i+1} \) where \( 0 \leq i < n \) do in parallel

\[
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(i) & \rightarrow N_i \\
c & = i \text{ and } c \rightarrow C_i
\end{align*}
\]

repeat \( \lceil \log^*(n) \rceil + 2 \) times

\[
\begin{align*}
C_{N_i} & \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k) \% 2) \neq ((c' \gg k) \% 2). \\
c & = 2 \cdot k + ((c \gg k) \% 2). \\
c & \rightarrow C_i
\end{align*}
\]

for \( r := 5 \) downto 3 do:

if \( c = r \) then

\[
\begin{align*}
C_{N_i} & \rightarrow c' \\
c' & \rightarrow C_i \\
C_{N_i} & \rightarrow c'' \\
c & := \min(\{0, 1, 2\} \setminus \{c', c''\}) \\
c & \rightarrow C_i
\end{align*}
\]
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-tree

for all \( P_{i+1} \) where \( 0 \leq i < n \) do in parallel

\[
\pi(i - 1) \rightarrow R_i \\
\pi(i) \rightarrow N_i \\
c = i \text{ and } c \rightarrow C_i
\]

repeat \( \lceil \log^*(n) \rceil + 2 \) times, if \( R_i \neq N_i \)

\[
C_{N_i} \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k) \% 2) \neq ((c' \gg k) \% 2). \\
c = 2 \cdot k + ((c \gg k) \% 2). \\
c \rightarrow C_i
\]

for \( r := 5 \) downto 3 do:

if \( c = r \) then

\[
C_{N_i} \rightarrow c' \\
\rightarrow C_i \\
C_{N_i} \rightarrow c'' \\
c := \min(\{0, 1, 2\} \setminus \{c', c''\}) \\
c \rightarrow C_i
\]
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

\textbf{for all} \ P_{i+1} \ \text{where} \ 0 \leq i < n \ \text{do in parallel}

\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(j - 1) & \rightarrow N_i \text{ with } j \text{ is father of } i \\
c & = i \text{ and } c \rightarrow C_i
\end{align*}

\textbf{repeat} \ \lceil \log^*(n) \rceil + 2 \ \text{times}

\begin{align*}
C_{N_i} & \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k) \% 2) & \neq ((c' \gg k) \% 2). \\
c & = 2 \cdot k + ((c \gg k) \% 2). \\
c & \rightarrow C_i
\end{align*}
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

$\pi(i - 1) \rightarrow R_i$
$\pi(j - 1) \rightarrow N_i$ with $j$ is father of $i$
$c = i$ and $c \rightarrow C_i$

repeat $\lceil \log^*(n) \rceil + 2$ times

$C_{N_i} \rightarrow c'$
minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.
$c = 2 \cdot k + ((c \gg k) \% 2)$.
$c \rightarrow C_i$

if $R_i = N_i$ then $c = \min(\{0, 1\} \setminus R_i)$ else $c = C_{N_i}$
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle
for all $P_{i+1}$ where $0 \leq i < n$ do in parallel
\begin{align*}
\pi(i - 1) & \rightarrow R_i \\
\pi(j - 1) & \rightarrow N_i \text{ with } j \text{ is father of } i \\
c & = i \text{ and } c \rightarrow C_i
\end{align*}
repeat $\lceil \log^*(n) \rceil + 2$ times
\begin{align*}
C_{N_i} & \rightarrow c' \\
\text{minimal } k \text{ with: } ((c \gg k)\%2) \neq ((c' \gg k)\%2). \\
c & = 2 \cdot k + ((c \gg k)\%2). \\
c & \rightarrow C_i \\
\text{if } R_i = N_i \text{ then } c = \min(\{0,1\} \setminus R_i) \text{ else } c = C_{N_i} \\
c & \rightarrow C_i
\end{align*}
Parallel Coloring Algorithm of (on) a tree (Idea)

Programm: color-cycle

for all $P_{i+1}$ where $0 \leq i < n$ do in parallel

\[
\pi(i - 1) \rightarrow R_i
\]

\[
\pi(j - 1) \rightarrow N_i \text{ with } j \text{ is father of } i
\]

\[
c = i \text{ and } c \rightarrow C_i
\]

repeat $\lceil \log^*(n) \rceil + 2$ times

\[
C_{N_i} \rightarrow c'
\]

minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$.

\[
c = 2 \cdot k + ((c \gg k) \% 2).
\]

\[
c \rightarrow C_i
\]

if $R_i = N_i$ then $c = \min(\{0, 1\} \setminus R_i)$ else $c = C_{N_i}$

\[
c \rightarrow C_i
\]

for $r := 5$ downto 3 do:

if $c = r$ then

\[
C_{N_i} \rightarrow c'
\]

\[
c' \rightarrow C_i
\]

\[
C_{N_i} \rightarrow c''
\]

\[
c := \min(\{0, 1, 2\} \setminus \{c', c''\})
\]

\[
c \rightarrow C_i
\]
Coloring a Tree

Theorem:
A tree with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.
Coloring a Tree

Theorem:

A tree with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.

Theorem:

A tree of $n$ processors may color itself in time $O(\log^* n)$ with at most 3 colors.

Proof: see above.
Eulerian cycle

**Definition:**

A graph $G = (V, E)$ is called Eulerian, iff there exists a cycle which visits each edge precisely once.
Eulerian cycle

**Definition:**
A graph $G = (V, E)$ is called Eulerian, iff there exists a cycle which visits each edge precisely once.

**Theorem**
A non-directed graph $G = (V, E)$ is Eulerian
- $G$ is connected and
- each node of $G$ has even degree.
Eulerian cycle

**Definition:**
A graph $G = (V, E)$ is called Eulerian, iff there exists a cycle which visits each edge precisely once.

**Theorem**
A non-directed graph $G = (V, E)$ is Eulerian
- $G$ is connected and
- each node of $G$ has even degree.

**Theorem**
A directed graph $G = (V, E)$ is Eulerian
- $G$ is strongly connected and
- each node has as many incoming edges as outgoing ones.
Eulerian cycle

Definition:
A graph $G = (V, E)$ is called Eulerian, iff there exists a cycle which visits each edge precisely once.

Theorem
A non-directed graph $G = (V, E)$ is Eulerian
- $G$ is connected and
- each node of $G$ has even degree.

Theorem
A directed graph $G = (V, E)$ is Eulerian
- $G$ is strong connected and
- each node has as many incoming edges as outgoing ones.

Problem: Compute Eulerian cycle on Eulerian graphs.
## Idea

- **Non Parallel:**

  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.
  - If $G$ is non-directed, then make a directed version of $G$.
  - Compute a cover of cycles.
  - Compute an additional cycle which meets each cycle precisely once.
  - Uses these to compute a cycle for $G$.
  - Delete some edges to get a Eulerian cycle for $G$. 

---

*Walter Unger 29.11.2016 20:11  WS2016/17*
### Idea

- **Non Parallel:**
  - Start with a free edge and follow free/unused edges till a cycle is closed.
Idea

- Non Parallel:
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
Idea

- Non Parallel:
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are is some cycle.
  - Join pairs of cycles into a single one.
Idea

- Non Parallel:
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.
Idea

- Non Parallel:
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.

- If \( G \) is non-directed, then make a directed version of \( G \).
Idea

- **Non Parallel:**
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.

- If $G$ is non-directed, then make a directed version of $G$.
- Compute a cover of cycles.
Idea

- Non Parallel:
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.
- If $G$ is non-directed, then make a directed version of $G$.
- Compute a cover of cycles.
- Compute an additional cycle which meets each cycle precisely once.
Idea

- **Non Parallel:**
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.

- If $G$ is non-directed, then make a directed version of $G$.
- Compute a cover of cycles.
- Compute an additional cycle which meets each cycle precisely once.
- **Uses these to compute a cycle for $G$**
Idea

- **Non Parallel:**
  - Start with a free edge and follow free/unused edges till a cycle is closed.
  - Repeat till all edges are in some cycle.
  - Join pairs of cycles into a single one.
  - Repeat till just one cycle remains.

- If $G$ is non-directed, then make a directed version of $G$.
- Compute a cover of cycles.
- Compute an additional cycle which meets each cycle precisely once.
- Uses these to compute a cycle for $G$
- **Delete some edges to get a Eulerian cycle for $G$.**
Change a non-directed Graph into a directed one

- $G$ contain $m$ non-directed edges.
Colorings I
3:16

Eulerian cycle

Introduction

Matchings

Colorings II

2/8

MIS

Coloring III

Walter Unger 29.11.2016 20:11

Change a non-directed Graph into a directed one
G contain m non-directed edges.
Substitute each non-directed edge with two directed ones:
{i, j} becomes (i, j) and (j, i).

Z

Simulations

WS2016/17


Change a non-directed Graph into a directed one

- G contains m non-directed edges.
- Substitute each non-directed edge with two directed ones: \{i, j\} becomes (i, j) and (j, i).
- Define a successor for each edge:
Change a non-directed Graph into a directed one

- G contain $m$ non-directed edges.
- Substitute each non-directed edge with two directed ones: 
  $\{i, j\}$ becomes $(i, j)$ and $(j, i)$.
- Define a successor for each edge:
  - The neighbors of $v$ are: $v_0, v_1, \ldots, v_{d-1}$. 
Change a non-directed Graph into a directed one

- $G$ contain $m$ non-directed edges.
- Substitute each non-directed edge with two directed ones: 
  \{i, j\} becomes (i, j) and (j, i).
- Define a successor for each edge:
  - The neighbors of $v$ are: $v_0, v_1, \cdots, v_{d-1}$.
  - Then define for all $i$:
    
    $Succ((v_i, v)) := (v, v_{(i+1) \mod d})$ und
    $Succ((v_{(i+1) \mod d}, v)) := (v, v_i)$. 
Change a non-directed Graph into a directed one

- \( G \) contain \( m \) non-directed edges.
- Substitute each non-directed edge with two directed ones: \( \{i, j\} \) becomes \((i, j)\) and \((j, i)\).
- Define a successor for each edge:
  - The neighbors of \( v \) are: \( v_0, v_1, \ldots, v_{d-1} \).
  - Then define for all \( i \):
    \[
    \text{Succ}((v_i, v)) := (v, v_{(i+1) \mod d}) \quad \text{and} \quad \text{Succ}((v_{(i+1) \mod d}, v)) := (v, v_i).
    \]
- Each directed edge is in precisely one cycle (defined by \( \text{Succ} \)).
Change a non-directed Graph into a directed one

- $G$ contain $m$ non-directed edges.
- Substitute each non-directed edge with two directed ones: 
  \[ \{i, j\} \text{ becomes } (i, j) \text{ and } (j, i) \].
- Define a successor for each edge:
  - The neighbors of $v$ are: $v_0, v_1, \cdots, v_{d-1}$.
  - Then define for all $i$:
    \[
    \text{Succ}((v_i, v)) := (v, v_{(i+1) \mod d}) \text{ und } \text{Succ}((v_{(i+1) \mod d}, v)) := (v, v_i).
    \]
- Each directed edge is in precisely one cycle (defined by Succ).
- For each cycle $C$ exists one cycle $C'$, which consists the reverse edges.
Change a non-directed Graph into a directed one

• $G$ contain $m$ non-directed edges.

• Substitute each non-directed edge with two directed ones: 
  \{i, j\} becomes $(i, j)$ and $(j, i)$.

• Define a successor for each edge:
  - The neighbors of $v$ are: $v_0, v_1, \cdots, v_{d-1}$.
  - Then define for all $i$:
    \[ \text{Succ}((v_i, v)) := (v, v_{(i+1) \mod d}) \text{ und} \]
    \[ \text{Succ}((v_{(i+1) \mod d}, v)) := (v, v_i). \]

• Each directed edge is in precisely one cycle (defined by $\text{Succ}$).

• For each cycle $C$ exists one cycle $C'$, which consists the reverse edges.

• We will now delete one of the two cycles $C$ or $C'$. 
Generating a directed Graph

- Identify the generated cycles:
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \min(((i, j), (k, l))) := \begin{cases} (i, j) & \text{if } i \leq k \lor i = k \land j < l \\ (k, l) & \text{otherwise} \end{cases} \).
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \text{min}((i,j), (k,l)) := \begin{cases} (i,j) & \text{if } i \leq k \lor i = k \land j < l \\ (k,l) & \text{otherwise} \end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \min(((i,j),(k,l))) := \begin{cases} (i,j) & \text{if } i \leq k \lor i = k \land j < l \\ (k,l) & \text{otherwise} \end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
  - For all edges \( e \) repeat \( \log m \) times:
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \min(((i, j), (k, l)) := \begin{cases} (i, j) & \text{if } i \leq k \lor i = k \land j < l \\ (k, l) & \text{otherwise} \end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
  - For all edges \( e \) repeat \( \log m \) times:
    - \( \text{Edge}'(e) = \min(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e))) \)
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \text{min}(((i, j), (k, l)) := \begin{cases} (i, j) & \text{if } i \leq k \lor i = k \land j < l \\ (k, l) & \text{otherwise} \end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
  - For all edges \( e \) repeat \( \log m \) times:
    - \( \text{Edge}'(e) = \text{min}(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e))) \)
    - \( \text{Succ}(e) = \text{Succ}(\text{Succ}(e)) \).
Generating a directed Graph

- Identify the generated cycles:
  - Let $\text{min}((i, j), (k, l)) := \begin{cases} (i, j) & \text{if } i \leq k \lor i = k \land j < l \\ (k, l) & \text{otherwise} \end{cases}$.
  - For each edge $e$ define $\text{Edge}'(e) = e$;
  - For all edges $e$ repeat $\log m$ times:
    - $\text{Edge}'(e) = \text{min}(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e)))$
    - $\text{Succ}(e) = \text{Succ}(\text{Succ}(e))$.
  - For each edge $(i, j)$: if $\text{min}((i, j), (j, i)) \neq (i, j)$ then let $\text{Edge}'(e) = 0$. 

Thus we have selected for each non-directed edge a directed one (resp. a direction).

Possible with $m$ in time $O(\log m)$. We consider in the following on directed graphs.
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \min((i,j),(k,l)) := \begin{cases} 
  (i,j) & \text{if } i \leq k \lor i = k \land j < l \\
  (k,l) & \text{otherwise}
\end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
  - For all edges \( e \) repeat \( \log m \) times:
    - \( \text{Edge}'(e) = \min(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e))) \)
    - \( \text{Succ}(e) = \text{Succ}(\text{Succ}(e)) \).
  - For each edge \( (i,j) \): if \( \min((i,j),(j,i)) \neq (i,j) \) then let \( \text{Edge}'(e) = 0 \).
  - Thus we have selected for each non-directed edge a directed one (resp. a direction).
Identify the generated cycles:

- Let \( \min(((i, j), (k, l))) := \begin{cases} (i, j) & \text{if } i \leq k \lor i = k \land j < l \\ (k, l) & \text{otherwise} \end{cases} \).
- For each edge \( e \) define \( \text{Edge}'(e) = e \);
- For all edges \( e \) repeat \( \log m \) times:
  - \( \text{Edge}'(e) = \min(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e))) \)
  - \( \text{Succ}(e) = \text{Succ}(\text{Succ}(e)) \).
- For each edge \( (i, j) \): if \( \min(((i, j), (j, i))) \neq (i, j) \) then let \( \text{Edge}'(e) = 0 \).
- Thus we have selected for each non-directed edge a directed one (resp. a direction).
- Possible with \( m \) in time \( O(\log m) \).
Generating a directed Graph

- Identify the generated cycles:
  - Let \( \min(((i,j), (k,l))) := \begin{cases} (i,j) & \text{if } i \leq k \lor i = k \land j < l \\ (k,l) & \text{otherwise} \end{cases} \).
  - For each edge \( e \) define \( \text{Edge}'(e) = e \);
  - For all edges \( e \) repeat \( \log m \) times:
    - \( \text{Edge}'(e) = \min(\text{Edge}'(e), \text{Edge}'(\text{Succ}(e))) \)
    - \( \text{Succ}(e) = \text{Succ}(\text{Succ}(e)) \).
  - For each edge \( (i,j) \): if \( \min(((i,j), (j,i))) \neq (i,j) \) then let \( \text{Edge}'(e) = 0 \).
  - Thus we have selected for each non-directed edge a directed one (resp. a direction).
  - Possible with \( m \) in time \( O(\log m) \).
  - We consider in the following on directed graphs.
Step 1

- Let $G = (V, E)$ be a directed graph.
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $Edge$.
  using the order: $(i, j) < (k, l) \iff j < l \lor (j = l \land i < k)$.
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $Edge$.
  using the order: $(i, j) < (k, l) \iff j < l \lor (j = l \land i < k)$.
- Sort the edges into an array $Succ$.
  using the order: $(i, j) < (k, l) \iff i < k \lor (i = k \land j < l)$. 
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $\text{Edge}$.
  using the order: $(i, j) < (k, l) \iff j < l \lor (j = l \land i < k)$.
- Sort the edges into an array $\text{Succ}$.
  using the order: $(i, j) < (k, l) \iff i < k \lor (i = k \land j < l)$.
- We have already defined the cycles:
  Successor of edge $e = \text{Edge}(i)$ is the edge $\text{Succ}(i)$. 
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $\text{Edge}$.
  using the order: \((i, j) < (k, l) \iff j < l \lor (j = l \land i < k)\).
- Sort the edges into an array $\text{Succ}$.
  using the order: \((i, j) < (k, l) \iff i < k \lor (i = k \land j < l)\).
- We have already defined the cycles:
  Successor of edge $e = \text{Edge}(i)$ is the edge $\text{Succ}(i)$.
- We also store in $P(i)$ the position of $\text{Succ}(i)$ in $\text{Edge}$. 
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $Edge$.
  using the order: $(i, j) < (k, l) \Leftrightarrow j < l \lor (j = l \land i < k)$.
- Sort the edges into an array $Succ$.
  using the order: $(i, j) < (k, l) \Leftrightarrow i < k \lor (i = k \land j < l)$.
- We have already defined the cycles:
  Successor of edge $e = Edge(i)$ is the edge $Succ(i)$.
- We also store in $P(i)$ the position of $Succ(i)$ in $Edge$.
- I.e. $Edge(P(i)) = Succ(i)$. 
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $\text{Edge}$ using the order: $(i, j) < (k, l) \iff j < l \lor (j = l \land i < k)$.
- Sort the edges into an array $\text{Succ}$ using the order: $(i, j) < (k, l) \iff i < k \lor (i = k \land j < l)$.
- We have already defined the cycles: Successor of edge $e = \text{Edge}(i)$ is the edge $\text{Succ}(i)$.
- We also store in $P(i)$ the position of $\text{Succ}(i)$ in $\text{Edge}$.
- I.e. $\text{Edge}(P(i)) = \text{Succ}(i)$.
- This information could be updated during the sorting of $\text{Succ}$.
Step 1

- Let $G = (V, E)$ be a directed graph.
- Sort the edges into an array $Edge$ using the order: $(i, j) < (k, l) \iff j < l \lor (j = l \land i < k)$.
- Sort the edges into an array $Succ$ using the order: $(i, j) < (k, l) \iff i < k \lor (i = k \land j < l)$.
- We have already defined the cycles:
  Successor of edge $e = Edge(i)$ is the edge $Succ(i)$.
- We also store in $P(i)$ the position of $Succ(i)$ in $Edge$.
  I.e. $Edge(P(i)) = Succ(i)$.
- This information could be updated during the sorting of $Succ$.
- This could be done in time $O(\log m)$ using $O(m)$ processors.
Step 2

- **Situation:** We have a directed graph covered by cycles.
Step 2

- **Situation:** We have a directed graph covered by cycles.
- **Problem:** Compute for each edge $e$ the cycles where $e$ belongs to.
Step 2

- **Situation:** We have a directed graph covered by cycles.

- **Problem:** Compute for each edge $e$ the cycles where $e$ belongs to.

- **Solution:** Compute for each cycle the minimal edge $((i, j) < (k, l) \iff i < k \lor (i = k \land j < l))$. 
Step 2

- **Situation:** We have a directed graph covered by cycles.
- **Problem:** Compute for each edge \( e \) the cycles where \( e \) belongs to.
- **Solution:** compute for each cycle the minimal edge \( ((i, j) < (k, l) \iff i < k \lor (i = k \land j < l)) \).
- **Algorithm:**
Step 2

- **Situation:** We have a directed graph covered by cycles.
- **Problem:** Compute for each edge $e$ the cycles where $e$ belongs to.
- **Solution:** Compute for each cycle the minimal edge $((i, j) < (k, l) \iff i < k \lor (i = k \land j < l))$.
- **Algorithm:**

  Program:
  
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  
  $CycleRep(i) := Succ(i)$
  
  for $i := 1$ to $\lceil \log m \rceil$ do:
  
  $CycleRep(i) := \min(CycleRep(i), CycleRep(P(i)))$
  
  $P(i) := P(P(i))$
Step 2

- **Situation:** We have a directed graph covered by cycles.
- **Problem:** Compute for each edge $e$ the cycles where $e$ belongs to.
- **Solution:** Compute for each cycle the minimal edge $((i, j) < (k, l) \iff i < k \lor (i = k \land j < l))$.
- **Algorithm:**

  Programm:
  
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  
  $\text{CycleRep}(i) := \text{Succ}(i)$
  
  for $i := 1$ to $\lceil \log m \rceil$ do:
  
  $\text{CycleRep}(i) := \min(\text{CycleRep}(i), \text{CycleRep}(P(i)))$
  
  $P(i) := P(P(i))$

- We use again the doubling technique.
Step 2

- Situation: We have a directed graph covered by cycles.
- Problem: Compute for each edge \( e \) the cycles where \( e \) belongs to.
- Solution: compute for each cycle the minimal edge \((i, j) < (k, l) \Leftrightarrow i < k \lor (i = k \land j < l)\).
- Algorithm:

Programm:

```plaintext
for all \( P_i \) where \( 1 \leq i \leq m \) do in parallel
    \( CycleRep(i) := Succ(i) \)
    for \( i := 1 \) to \( \lceil \log m \rceil \) do:
        \( CycleRep(i) := \min(CycleRep(i), CycleRep(P(i))) \)
        \( P(i) := P(P(i)) \)
```

- We use again the doubling technique.
- Possible in time \( O(\log m) \) using \( O(m) \) Processors.
Step 2 (Continued)

- **Situation:** the cycles of the coverage are identified by *CycleRep.*
Step 2 (Continued)

- Situation: the cycles of the coverage are identified by $CycleRep$.
- Problem: join the cycle into a single one.
Step 2 (Continued)

- **Situation:** the cycles of the coverage are identified by $CycleRep$.
- **Problem:** join the cycle into a single one.
- **Solution:** Identify the nodes of the cycle.
Step 2 (Continued)

- Situation: the cycles of the coverage are identified by $CycleRep$.
- Problem: join the cycle into a single one.
- Solution: Identify the nodes of the cycle.
- $C = \{ CycleRep(i) \mid 1 \leq i \leq m \}$. (Note $C$ is a edge set)
Step 2 (Continued)

- **Situation:** the cycles of the coverage are identified by \( \text{CycleRep} \).
- **Problem:** join the cycle into a single one.
- **Solution:** Identify the nodes of the cycle.
- \( C = \{ \text{CycleRep}(i) \mid 1 \leq i \leq m \} \). (Note \( C \) is a edge set)
- \( G' = V \cup C \)
Situation: the cycles of the coverage are identified by $\text{CycleRep}$.

Problem: join the cycle into a single one.

Solution: Identify the nodes of the cycle.

$C = \{ \text{CycleRep}(i) \mid 1 \leq i \leq m \}$. (Note $C$ is a edge set)

$G' = V \cup C$

$E' = \{(u,v) \mid u \in V, v \in C : v \text{ is identified in the cycle by } u\}$
Step 2 (Continued)

- Situation: the cycles of the coverage are identified by \( \text{CycleRep} \).
- Problem: join the cycle into a single one.
- Solution: Identify the nodes of the cycle.

\[ C = \{ \text{CycleRep}(i) \mid 1 \leq i \leq m \} \]  
(Note \( C \) is a edge set)

\[ G' = V \cup C \]

\[ E' = \{ (u, v) \mid u \in V, v \in C : v \text{ is identified in the cycle by } u \} \]

Computing of \( E' \):
Step 2 (Continued)

- Situation: the cycles of the coverage are identified by $CycleRep$.
- Problem: join the cycle into a single one.
- Solution: Identify the nodes of the cycle.
  
  $C = \{CycleRep(i) | 1 \leq i \leq m\}$. (Note $C$ is a edge set)
  
  $G' = V \cup C$
  
  $E' = \{(u, v) | u \in V, v \in C : v \text{ is identified in the cycle by } u\}$
  
  Computing of $E'$:

  Programm:

  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  
  $(u, v) = Edge(i)$
  
  $Edge'(2 \cdot i) = (u, CycleRep(i))$
  
  $Edge'(2 \cdot i + 1) = (v, CycleRep(i))$
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
Step 2 (Continued)

- **Situation:** Cover of cycles and graph $G'$ defined.
- **Problem:** there are multiple edges.
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort Edge'.

Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.
- Programm:
  
  ```
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$
  ```
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.

Programm:

```
for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$
```

- Sort $Edge'$. 
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.
- Program:
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$
- Sort $Edge'$.
- Consider only the first $|E'|$ elements of $Edge'$. 
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.
- 
  **Programm:**

  
  for all $P_i$ where $1 \leq i \leq m$ do in parallel 
  
  if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$

- Sort $Edge'$.

- Consider only the first $|E'|$ elements of $Edge'$.

- Problem: node $u$ could appear several times in a cycle $v$. 
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.

Programm:

\[
\text{for all } P_i \text{ where } 1 \leq i \leq m \text{ do in parallel}
\]

\[
\text{if } Edge'(i) = Edge'(i + 1) \text{ then } Edge(i) = \infty
\]

- Sort $Edge'$.

- Consider only the first $|E'|$ elements of $Edge'$.
- Problem: node $u$ could appear several times in a cycle $v$.
- As before we may compute a single representative.
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.
- Program:
  
  ```
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$
  ```
- Sort $Edge'$.
- Consider only the first $|E'|$ elements of $Edge'$.
- Problem: node $u$ could appear several times in a cycle $v$.
- As before we may compute a single representative.
- Let these edge be $(i, u) = Cert(u, v)$. 
Step 2 (Continued)

- Situation: Cover of cycles and graph $G'$ defined.
- Problem: there are multiple edges.
- Solution: sort them out.
- Sort $Edge'$.

**Programm:**

```plaintext
for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Edge'(i) = Edge'(i + 1)$ then $Edge(i) = \infty$
```

- Sort $Edge'$.
- Consider only the first $|E'|$ elements of $Edge'$.
- Problem: node $u$ could appear several times in a cycle $v$.
- As before we may compute a single representative.
- Let these edge be $(i, u) = Cert(u, v)$.
- May be done in time $O(\log m)$ using $O(m)$ processors.
Step 3

- Situation: Covering of the cycles and graph \( G' \) computed.
Step 3

- **Situation:** Covering of the cycles and graph $G'$ computed.
- **Problem:** Compute cycle in $G'$. 
Step 3

- **Situation**: Covering of the cycles and graph $G'$ computed.
- **Problem**: Compute cycle in $G'$.
- **Solution**: compute spanning tree $T$ for the bipartite Graph $G'$. 
Step 3

- **Situation:** Covering of the cycles and graph $G'$ computed.
- **Problem:** Compute cycle in $G'$.
- **Solution:** compute spanning tree $T$ for the bipartite Graph $G'$.
- **To compute spanning tree we need $O(\log^2 m)$ time with $O(m/\log^2 m)$ Processors.**
Step 3

- **Situation**: Covering of the cycles and graph $G'$ computed.
- **Problem**: Compute cycle in $G'$.
- **Solution**: compute spanning tree $T$ for the bipartite Graph $G'$.
- To compute spanning tree we need $O(\log^2 m)$ time with $O(m/\log^2 m)$ Processors.
- Then we substitute each edge in $T$ with two directed edges.
Situation: Covering of the cycles and graph $G'$ computed.

Problem: Compute cycle in $G'$.

Solution: Compute spanning tree $T$ for the bipartite Graph $G'$.

To compute spanning tree we need $O(\log^2 m)$ time with $O(m/\log^2 m)$ Processors.

Then we substitute each edge in $T$ with two directed edges.

The new graph $T'$ is Eulerian.
Step 3

- **Situation:** Covering of the cycles and graph $G'$ computed.
- **Problem:** Compute cycle in $G'$.
- **Solution:** compute spanning tee $T$ for the bipartite Graph $G'$.
- To compute spanning tree we need $O(\log^2 m)$ time with $O(m/\log^2 m)$ Processors.
- Then we substitute each edge in $T$ with two directed edges.
- The new graph $T'$ is Eulerian.
- **The Eulerian cycle is easy to find:**
Step 3

- Situation: Covering of the cycles and graph $G'$ computed.
- Problem: Compute cycle in $G'$.
- Solution: compute spanning tree $T$ for the bipartite Graph $G'$.
- To compute spanning tree we need $O(\log^2 m)$ time with $O(m/\log^2 m)$ Processors.
- Then we substitute each edge in $T$ with two directed edges.
- The new graph $T'$ is Eulerian.
- The Eulerian cycle is easy to find:
  - To do so, compute for each node of the tree the order of edges.
Step 3

- Situation: Covering of the cycles and graph \( G' \) computed.
- Problem: Compute cycle in \( G' \).
- Solution: compute spanning tree \( T \) for the bipartite Graph \( G' \).
- To compute spanning tree we need \( O(\log^2 m) \) time with \( O(m/\log^2 m) \) Processors.
- Then we substitute each edge in \( T \) with two directed edges.
- The new graph \( T' \) is Eulerian.
- The Eulerian cycle is easy to find:
- To do so, compute for each node of the tree the order of edges.
- Could be done in time \( O(\log m) \) using \( O(m) \) processors.
Step 4

- **Situation:** We have a cover of cycles for $G$ and $T'$. 
Step 4

- **Situation:** We have a cover of cycles for $G$ and $T$.
- **Problem:** Find cycle $L$ in $G'$. 

For each cycle $v$ in $G$, $\text{Cert}(u, v)$ gives us an edge, at which we may exchange between $v$ and the cycle in $T'$. These points of change will be used to construct a single cycle $L$. Time $O(1)$ using $O(m)$ Processors.
Step 4

- **Situation:** We have a cover of cycles for $G$ and $T'$.
- **Problem:** Find cycle $L$ in $G'$.
- **Solution:** Combine the cycles using $Cert(u, v)$. 
Step 4

- **Situation**: We have a cover of cycles for $G$ and $T'$. 
- **Problem**: Find cycle $L$ in $G'$.  
- **Solution**: Combine the cycles using $\text{Cert}(u, v)$. 
- $L$ will also contain the Eulerian cycle in $G$. 

Step 4

- **Situation:** We have a cover of cycles for $G$ and $T'$.
- **Problem:** Find cycle $L$ in $G'$.
- **Solution:** Combine the cycles using $Cert(u, v)$.
- $L$ will also contain the Eulerian cycle in $G$.
- For each cycle $v$ in $G$ $Cert(u, v)$ gives us an edge, at which we may exchange between $v$ and the cycle in $T'$.  

Step 4

- **Situation:** We have a cover of cycles for $G$ and $T'$.

- **Problem:** Find cycle $L$ in $G'$.

- **Solution:** Combine the cycles using $\text{Cert}(u, v)$.

- $L$ will also contain the Eulerian cycle in $G$.

- For each cycle $v$ in $G$ $\text{Cert}(u, v)$ gives us an edge, at which we may exchange between $v$ and the cycle in $T'$.

- These points of change will be used to construct a single cycle $L$. 
Step 4

- **Situation:** We have a cover of cycles for $G$ and $T'$.
- **Problem:** Find cycle $L$ in $G'$.
- **Solution:** Combine the cycles using $Cert(u, v)$.
- $L$ will also contain the Eulerian cycle in $G$.
- For each cycle $v$ in $G$ $Cert(u, v)$ gives us an edge, at which we may exchange between $v$ and the cycle in $T'$.
- These points of change will be used to construct a single cycle $L$.
- **Time $O(1)$ using $O(m)$ Processors.**
Step 5

- **Situation:** we have a cycle for $G$ and $T'$. 
Step 5

- **Situation:** we have a cycle for $G$ and $T'$.
- **Problem:** find cycle in $G$. 

Program:

```
for all $P_i$ where $1 \leq i \leq m$ do in parallel
if $\text{Succ}(i) \in T'$ then
    $\text{Succ}(i) := \text{Succ}(\text{Succ}(i))$
if $\text{Succ}(i) \in T'$ then
    $\text{Succ}(i) := \text{Succ}(\text{Succ}(i))$
```

Uses time $O(1)$ with $O(m)$ processors.

Total time is: $O(\log_2 m)$ using $O(m)$ processors.

Also possible: $O(\log_2 m)$ time using $O(m/\log_2 m)$ processors.
Step 5

- **Situation:** we have a cycle for $G$ and $T'$.
- **Problem:** find cycle in $G$.
- **Solution:** delete edges from $T'$.
Step 5

- **Situation:** we have a cycle for $G$ and $T'$.
- **Problem:** find cycle in $G$.
- **Solution:** delete edges from $T'$. 
Step 5

- **Situation**: we have a cycle for $G$ and $T'$.
- **Problem**: find cycle in $G$.
- **Solution**: delete edges from $T'$.
- **Programm**:

  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  
  if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
  
  if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
Step 5

- Situation: we have a cycle for $G$ and $T'$.
- Problem: find cycle in $G$.
- Solution: delete edges from $T'$.
- Program:
  
  ```plaintext
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
    if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
  ```

- Uses time $O(1)$ with $O(m)$ processors.
Step 5

• Situation: we have a cycle for $G$ and $T'$.
• Problem: find cycle in $G$.
• Solution: delete edges from $T'$.
• Programm:
  
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
    if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
    if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$

• Uses time $O(1)$ with $O(m)$ processors.
• Total time is: $O(\log^2 m)$ using $O(m)$ processors.
Step 5

- **Situation**: we have a cycle for $G$ and $T$.
- **Problem**: find cycle in $G$.
- **Solution**: delete edges from $T$.
- **Programm**:
  
  ```
  for all $P_i$ where $1 \leq i \leq m$ do in parallel
  
  if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
  if $Succ(i) \in T'$ then $Succ(i) := Succ(Succ(i))$
  ```

- Uses time $O(1)$ with $O(m)$ processors.
- Total time is: $O(\log^2 m)$ using $O(m)$ processors.
- Also possible: $O(\log^2 m)$ time using $O(m/\log^2 m)$ processors.
Definition

Let \( G = (V, E) \) be a non-directed graph.
Definition

- Let $G = (V, E)$ be a non-directed graph.
- $M \subset E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$. 
Definition

Let $G = (V, E)$ be a non-directed graph.

$M \subset E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.

$M$ is called maximal matching, iff $\nexists e \in E : M \cup \{e\}$ is a matching.
Definition

Let $G = (V, E)$ be a non-directed graph.

- $M \subset E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.
- $M$ is called maximal matching, iff $\nexists e \in E : M \cup \{e\}$ is a matching.
- $M$ is called maximum matching, iff for all matchings $M'$ we have $|M'| \leq |M|$.
Definition

- Let $G = (V, E)$ be a non-directed graph.
- $M \subseteq E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.
- $M$ is called maximal matching, iff $\not\exists e \in E : M \cup \{e\}$ is a matching.
- $M$ is called maximum matching, iff for all matchings $M'$ we have $|M'| \leq |M|$.

Sequential: $O(m \log m)$ for maximal matching.
### Definition

- Let $G = (V, E)$ be a non-directed graph.
- $M \subseteq E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.
- $M$ is called maximal matching, iff $\nexists e \in E : M \cup \{e\}$ is a matching.
- $M$ is called maximum matching, iff for all matchings $M'$ we have $|M'| \leq |M|$.

- Sequential: $O(m \log m)$ for maximal matching.
- Idea: Choose any free edge and delete all incident edges.
Definition

Let $G = (V, E)$ be a non-directed graph.

- $M \subseteq E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.
- $M$ is called maximal matching, iff $\not\exists e \in E : M \cup \{e\}$ is a matching.
- $M$ is called maximum matching, iff for all matchings $M'$ we have $|M'| \leq |M|$.

- Sequential: $O(m \log m)$ for maximal matching.
- Idea: Choose any free edge and delete all incident edges.
- Sequential: $O(m^3)$ for maximum matching.
Definition

Let $G = (V, E)$ be a non-directed graph.

- $M \subset E$ is called a matching, iff $\forall e, e' \in M : e \cap e' = \emptyset$.
- $M$ is called maximal matching, iff $\nexists e \in E : M \cup \{e\}$ is a matching.
- $M$ is called maximum matching, iff for all matchings $M'$ we have $|M'| \leq |M|$.

Sequential: $O(m \log m)$ for maximal matching.

Idea: Choose any free edge and delete all incident edges.

Sequential: $O(m^3)$ for maximum matching.

Idea: enlarging alternating pathes.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$. 

...
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{\frac{3}{2}} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$. 
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-th phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$. 
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$.
- Within each phase $F_i$ we will call the procedure $\text{DegreeSplit}$ $(1 + \log(\Delta(G)))$-times.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$.
- Within each phase $F_i$ we will call the procedure $DegreeSplit$ $(1 + \log(\Delta(G)))$-times.
- Within each step within a phase we will half the node degree.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$.
- Within each phase $F_i$ we will call the procedure $DegreeSplit$ $(1 + \log(\Delta(G)))$-times.
- Within each step within a phase we will half the node degree.
- We denote with $G(i,j)$ the graph considered in the $j$-th Step of the $i$-th phase.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$.
- Within each phase $F_i$ we will call the procedure $\text{DegreeSplit}$ $(1 + \log(\Delta(G)))$-times.
- Within each step within a phase we will half the node degree.
- We denote with $G(i, j)$ the graph considered in the $j$-th Step of the $i$-th phase.
- We will describe the procedure $\text{DegreeSplit}$.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\bigcup M_i$.
- Within each phase $F_i$ we will call the procedure $DegreeSplit$ $(1 + \log(\Delta(G)))$-times.
- Within each step within a phase we will half the node degree.
- We denote with $G(i, j)$ the graph considered in the $j$-th Step of the $i$-th phase.
- We will describe the procedure $DegreeSplit$.
- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1}$.
Idea

- Let $\Delta(G)$ be the maximal degree of $G$.
- Enlarge the matching step by step by several edges.
- There will be $O(\log_{3/2} n)$ phases.
- $i$-te phase $F_i$ has $G_i$ as input and will output $M_i$.
- $G_1 = G$ and final result: $\cup M_i$.
- Within each phase $F_i$ we will call the procedure $DegreeSplit$ $(1 + \log(\Delta(G)))$-times.
- Within each step within a phase we will half the node degree.
- We denote with $G(i, j)$ the graph considered in the $j$-th Step of the $i$-th phase.
- We will describe the procedure $DegreeSplit$.
- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1}$.
- We will call all nodes $v$ with $\delta(v) \geq 2^k$ active.
Step 1

1. Compute all active nodes of $G(i,j)$
Step 1

- Compute all active nodes of $G(i, j)$
  - Determine the degree in time $O(\log \Delta(G(i, j)))$ with $O(m)$ processors.
Step 1

- Compute all active nodes of $G(i, j)$
  - Determine the degree in time $O(\log \Delta(G(i, j)))$ with $O(m)$ processors.
  - Determine the maximum degree in time $O(\log n)$ with $O(n)$ processors.
Step 1

- Compute all active nodes of $G(i,j)$
  - Determine the degree in time $O(\log \Delta(G(i,j)))$ with $O(m)$ processors.
  - Determine the maximum degree in time $O(\log n)$ with $O(n)$ processors.
  - Then the active nodes are known in time $O(1)$ using $O(n)$ processors.
Step 1

- Compute all active nodes of $G(i,j)$
  - Determine the degree in time $O(\log \Delta(G(i,j)))$ with $O(m)$ processors.
  - Determine the maximum degree in time $O(\log n)$ with $O(n)$ processors.
  - Then the active nodes are known in time $O(1)$ using $O(n)$ processors.

- **Total running time:** $O(\log n)$ using $O(m)$ processors.
Step 2

- Compute the graph $G^*(i,j)$ as follows:
Step 2

- Compute the graph $G^*(i, j)$ as follows:
  - Compute all nodes that are incident to active nodes.
Step 2

- Compute the graph $G^*(i, j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
Step 2

- Compute the graph $G^*(i,j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
  - If there are nodes with odd degree connect them to a new node $v$. 
Step 2

- Compute the graph $G^*(i, j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
  - If there are nodes with odd degree connect them to a new node $v$.

- Total running time: $O(\log n)$ using $O(m)$ processors.
Step 2

- Compute the graph $G^*(i,j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
  - If there are nodes with odd degree connect them to a new node $v$.

- Total running time: $O(\log n)$ using $O(m)$ processors.

- $G^*(i,j)$ might not be connected.
Step 2

- Compute the graph $G^*(i,j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
  - If there are nodes with odd degree connect them to a new node $v$.

- Total running time: $O(\log n)$ using $O(m)$ processors.
- $G^*(i,j)$ might not be connected.
- Each component of $G^*(i,j)$ contains an Eulerian cycle.
Step 2

- Compute the graph $G^*(i,j)$ as follows:
  - Compute all nodes that are incident to active nodes.
  - Determine the new node degree.
  - If there are nodes with odd degree connect them to a new node $v$.

- Total running time: $O(\log n)$ using $O(m)$ processors.
- $G^*(i,j)$ might not be connected.
- Each component of $G^*(i,j)$ contains an Eulerian cycle.
- Note that each node $v$ has even degree.
Step 3

Compute an Eulerian cycle on each component of $G^*(i,j)$. 
Step 3

- Compute an Eulerian cycle on each component of $G^*(i, j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
Step 3

- Compute an Eulerian cycle on each component of $G^*(i,j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
Step 3

- Compute an Eulerian cycle on each component of $G^*(i,j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
- Label the edges from the Eulerian cycle alternating with 0 and 1.
Step 3

- Compute an Eulerian cycle on each component of $G^*(i,j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
- Label the edges from the Eulerian cycle alternating with 0 and 1.
- For the component with the additional node $v$ start with $v$ using label 0.
Step 3

- Compute an Eulerian cycle on each component of $G^*(i,j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
- Label the edges from the Eulerian cycle alternating with 0 and 1.
- For the component with the additional node $v$ start with $v$ using label 0.
- For all other components start at an arbitrary node with label 1.
Step 3

- Compute an Eulerian cycle on each component of $G^*(i,j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
- Label the edges from the Eulerian cycle alternating with 0 and 1.
- For the component with the additional node $v$ start with $v$ using label 0.
- For all other components start at an arbitrary node with label 1.
- Running time: $O(\log n)$ with $O(m + n)$ processors.
Step 3

- Compute an Eulerian cycle on each component of $G^* (i, j)$.
- This needs time $O(\log^2 n)$ with $O(m + n)$ processors.
- Note that the additional $n$ processors result from the additional edges.
- Label the edges from the Eulerian cycle alternating with 0 and 1.
- For the component with the additional node $v$ start with $v$ using label 0.
- For all other components start at an arbitrary node with label 1.
- Running time: $O(\log n)$ with $O(m + n)$ processors.
- Use Parallel Prefix to compute the labels.
Step 4

- Delete all edges with label 0.
Step 4

- Delete all edges with label 0.
- If the remaining graph $G^{**}(i, j)$ is not a matching then $G(i, j + 1) = G^{**}(i, j) \setminus \{v\}$.
Step 4

- Delete all edges with label 0.
- If the remaining graph $G^{**}(i, j)$ is not a matching then $G(i, j + 1) = G^{**}(i, j) \setminus \{v\}$.
- If the remaining graph $G^{**}(i, j)$ is a matching then $M_i = E(G^{**}(i, j))$. 
Step 4

- Delete all edges with label 0.
- If the remaining graph $G^{**}(i,j)$ is not a matching then 
  $G(i, j + 1) = G^{**}(i, j) \setminus \{v\}$.
- If the remaining graph $G^{**}(i, j)$ is a matching then $M_i = E(G^{**}(i, j))$.
- Running time: $O(1)$ with $O(m + n)$ processors.
Step 4

- Delete all edges with label 0.
- If the remaining graph \( G^{**}(i,j) \) is not a matching then 
  \[ G(i,j + 1) = G^{**}(i,j) \setminus \{v\}. \]
- If the remaining graph \( G^{**}(i,j) \) is a matching then \( M_i = E(G^{**}(i,j)) \).
- Running time: \( O(1) \) with \( O(m + n) \) processors.
- Running time of the procedure \( \text{DegreeSplit} \): \( O(\log^2 n) \) with \( O(m + n) \) processors.
Step 4

- Delete all edges with label 0.
- If the remaining graph $G^{**}(i, j)$ is not a matching then $G(i, j + 1) = G^{**}(i, j) \setminus \{v\}$.
- If the remaining graph $G^{**}(i, j)$ is a matching then $M_i = E(G^{**}(i, j))$.
- Running time: $O(1)$ with $O(m + n)$ processors.
- Running time of the procedure $DegreeSplit$: $O(\log^2 n)$ with $O(m + n)$ processors.
- It remains to show: After at most $1 + \log(\Delta(G(i, j)))$ steps $DegreeSplit$ computes a matching.
Step 4

- Delete all edges with label 0.
- If the remaining graph $G^{**}(i, j)$ is not a matching then $G(i, j + 1) = G^{**}(i, j) \setminus \{v\}$.
- If the remaining graph $G^{**}(i, j)$ is a matching then $M_i = E(G^{**}(i, j))$.
- Running time: $O(1)$ with $O(m + n)$ processors.
- Running time of the procedure $DegreeSplit$: $O(\log^2 n)$ with $O(m + n)$ processors.
- It remains to show: After at most $1 + \log(\Delta(G(i, j)))$ steps $DegreeSplit$ computes a matching.
- It remains to show: After at most $O(\log_{3/2} n)$ phases the matching is optimal.
Inner loop

Lemma:
Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.
Inner loop

Lemma:
Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:
- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$. 
Inner loop

**Lemma:**
Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

**Proof:**
- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Ket $G_1$ be the result of an iteration.
Lemma:

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
Lemma:

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
Lemma:

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lceil \delta_G(v)/2 \rceil + 1$. 
Inner loop

**Lemma:**
Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

**Proof:**
- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lceil \delta_G(v)/2 \rceil + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$. 
Lemma:

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lfloor \delta_G(v)/2 \rfloor + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$.
  - Then $v$ stays active in $G_1$. 

Lemma:

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lceil \delta_G(v)/2 \rceil + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$.
  - Then $v$ stays active in $G_1$.
- Hence the degree is halved in every step.
Inner loop

Lemma:

Let $G$ be the input of $\text{DegreeSplit}$, then $\text{DegreeSplit}$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

Proof:

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lfloor \delta_G(v)/2 \rfloor + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$.
  - Then $v$ stays active in $G_1$.
- Hence the degree is halved in every step.
- There exists a $k' \leq k$ such that $G_{k'}$ has a degree of 3.
Inner loop

**Lemma:**

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

**Proof:**

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Ket $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lfloor \delta_G(v)/2 \rfloor + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$.
- Then $v$ stays active in $G_1$.
- Hence the degree is halved in every step.
- There exists a $k' \leq k$ such that $G_{k'}$ has a degree of 3.
- After two more iterations the degree is at most one.
**Lemma:**

Let $G$ be the input of $DegreeSplit$, then $DegreeSplit$ will compute a matching after $1 + \log(\Delta(G))$ iterations.

**Proof:**

- Let $k$ be the smallest number with $2^k \leq \Delta(G) \leq 2^{k+1} + 1$.
- Let $G_1$ be the result of an iteration.
- Let $v$ be active in $G$. It holds:
  - $2^k \leq \delta_G(v)$.
  - $\lfloor \delta_G(v)/2 \rfloor \leq \delta_{G_1}(v) \leq \lfloor \delta_G(v)/2 \rfloor + 1$.
  - $2^{k-1} \leq \delta_{G_1}(v) \leq 2^k + 1$.
- Hence the degree is halved in every step.
- There exists a $k' \leq k$ such that $G_{k'}$ has a degree of 3.
- After two more iterations the degree is at most one.
- So a matching is found.
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.
Lemma:
A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$. 
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$. 
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$.
- ($C \subseteq V$ is a vertex cover if $\forall e \in E : e \cap C \neq \emptyset$)
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$.
- $(C \subset V$ is a vertex cover if $\forall e \in E : e \cap C \neq \emptyset)$
- We show the following;
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$.
- $(C \subseteq V$ is a vertex cover if $\forall e \in E : e \cap C \neq \emptyset)$
- We show the following;
  - Half of the nodes in a vertex cover $A_i$ can be made incident to edges from $M_i$. 
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$.

$(C \subseteq V$ is a vertex cover if $\forall e \in E : e \cap C \neq \emptyset)$

- We show the following:
  - Half of the nodes in a vertex cover $A_i$ can be made incident to edges from $M_i$.
  - This means it holds: $|A_i/G_{i+1}| \leq |A_i|/2$.
    with $A_i/G_{i+1}$ the nodes of $A_i$ in $G_{i+1}$
Lemma:

A logarithmic number of phases is enough to compute a maximum matching.

Proof:

- Let $A_i$ be the nodes that are active in phase $F_i$.
- Then $A_i$ is a vertex cover of $G_i$.
- ($C \subset V$ is a vertex cover if $\forall e \in E : e \cap C \neq \emptyset$)
- We show the following;
  - Half of the nodes in a vertex cover $A_i$ can be made incident to edges from $M_i$.
  - This means it holds: $|A_i/G_{i+1}| \leq |A_i|/2$.
    with $A_i/G_{i+1}$ the nodes of $A_i$ in $G_{i+1}$
  - There are vertex covers $C_i$: $|C_{i+1}| \leq 2 \cdot |C_i|/3$. 


Outer loop (Proof)

- Let $G_k = (V, E_k)$ be the graph in the third to last loop of $DegreeSplit$. 
Outer loop (Proof)

- Let $G_k = (V, E_k)$ be the graph in the third to last loop of $DegreeSplit$.
- W.l.o.g. $G_k$ is connected with degree $\leq 3$. 
Outer loop (Proof)

- Let $G_k = (V, E_k)$ be the graph in the third to last loop of $DegreeSplit$.
- W.l.o.g. $G_k$ is connected with degree $\leq 3$.
- $DegreeSplit$ can w.l.o.g. remove the smallest set of edges.
Outer loop (Proof)

- Let $G_k = (V, E_k)$ be the graph in the third to last loop of $DegreeSplit$.
- W.l.o.g. $G_k$ is connected with degree $\leq 3$.
- $DegreeSplit$ can w.l.o.g. remove the smallest set of edges.
- Hence it holds $|M_i| \geq |E_k|/4$. 


Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
  - Both end points of an edge from $A_i$ belong to $A_i$ and
Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
  - Both end points of an edge from $A_i$ belong to $A_i$ and
  - at least half of them are incident to $M_i$. 
Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
  - Both end points of an edge from $A_i$ belong to $A_i$ and
  - at least half of them are incident to $M_i$.

- If $|E_k| < |A_i|$ then $G_k$ is a tree.
Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
  - Both end points of an edge from $A_i$ belong to $A_i$ and
  - at least half of them are incident to $M_i$.

- If $|E_k| < |A_i|$ then $G_k$ is a tree.
  - We remove edges from $G_k$ that have a leaf as one of its end points.
Outer loop (Proof)

- If $|E_k| \geq |A_i|$ then $M_i$ contains at least $|A_i|/4$ edges.
  - Both end points of an edge from $A_i$ belong to $A_i$ and
  - at least half of them are incident to $M_i$.

- If $|E_k| < |A_i|$ then $G_k$ is a tree.
  - We remove edges from $G_k$ that have a leaf as one of its end points.
  - Furthermore the incident edges are removed.
Outer loop (Proof)

Because $\Delta(G_k) \leq 3$ at most 2 trees $T_1$ and $T_2$ remain (with $n_1 + n_2$ nodes).
Outer loop (Proof)

- Because $\Delta(G_k) \leq 3$ at most 2 trees $T_1$ and $T_2$ remain (with $n_1 + n_2$ nodes).
- Then $((n_1 - 1) + (n_2 - 1))/4$ edges are added to $M_i$. 
Outer loop (Proof)

- Because $\Delta(G_k) \leq 3$ at most 2 trees $T_1$ and $T_2$ remain (with $n_1 + n_2$ nodes).
- Then $((n_1 - 1) + (n_2 - 1))/4$ edges are added to $M_i$.
- Then $M_i$ contains $|A_i|/2$ nodes.
Outer loop (Proof)

- Because $\Delta(G_k) \leq 3$ at most 2 trees $T_1$ and $T_2$ remain (with $n_1 + n_2$ nodes).
- Then $((n_1 - 1) + (n_2 - 1))/4$ edges are added to $M_i$.
- Then $M_i$ contains $|A_i|/2$ nodes.
- Then it holds: $|A_i/G_{i+1}| \leq |A_i|/2$. 

$|M_i| \geq |E_k|/4$
We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{-1}|V|$.
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^i |V|$.

- We will show that $|C_{i+1}| \leq 2|C_i|/3$. 

\[
|M_i| \geq |E_k|/4 \\
|A_i/G_{i+1}| \leq |A_i|/2
\]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$. 

\[
|M_i| \geq |E_k|/4 \quad |A_i / G_{i+1}| \leq |A_i|/2
\]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$. 

$$|M_i| \geq |E_k|/4$$
$$|A_i/G_{i+1}| \leq |A_i|/2$$
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$. 

\[ |M_i| \geq |E_k|/4 \]
\[ |A_i/G_{i+1}| \leq |A_i|/2 \]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^i |V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.

We will show that $|C_{i+1}| \leq 2|C_i|/3$.

Basis: $i = 1$: Choose $C_1 = V$.

Case 1: $|A_i| \leq 4|C_i|/3$.

- In phase $i$ half of the nodes are removed from $A_i$.
- $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
- $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$. 

\[ |M_i| \geq |E_k|/4 \]
\[ |A_i/G_{i+1}| \leq |A_i|/2 \]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^i \leq |V|$.

- We will show that $|C_{i+1}| \leq 2|C_i|/3$.

- Basis: $i = 1$: Choose $C_1 = V$.

- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i / G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.

- Case 2: $|A_i| > 4|C_i|/3$. 
Outer loop (Proof)

We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.

We will show that $|C_{i+1}| \leq 2|C_i|/3$.

Basis: $i = 1$: Choose $C_1 = V$.

Case 1: $|A_i| \leq 4|C_i|/3$.
  
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.

Case 2: $|A_i| > 4|C_i|/3$.
  
  - Half of the nodes from $A_i$ are removed.
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^i|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.
- Case 2: $|A_i| > 4|C_i|/3$.
  - Half of the nodes from $A_i$ are removed.
  - These have end points in $M_i$. 

\[ |M_i| \geq |E_k|/4 \]
\[ |A_i/G_{i+1}| \leq |A_i|/2 \]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.
- Case 2: $|A_i| > 4|C_i|/3$.
  - Half of the nodes from $A_i$ are removed.
  - These have end points in $M_i$.
  - $C_i$ is a vertex cover of $G_i$. 

\[ |M_i| \geq |E_k|/4 \]
\[ |A_i/G_{i+1}| \leq |A_i|/2 \]
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.
- Case 2: $|A_i| > 4|C_i|/3$.
  - Half of the nodes from $A_i$ are removed.
  - These have end points in $M_i$.
  - $C_i$ is a vertex cover of $G_i$.
  - Then every edge has at least one end point in $C_i$. 

$|M_i| \geq |E_k|/4$
$|A_i/G_{i+1}| \leq |A_i|/2$
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.

Case 1: $|A_i| \leq 4|C_i|/3$.

- In phase $i$ half of the nodes are removed from $A_i$.
- $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
- $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.

Case 2: $|A_i| > 4|C_i|/3$.

- Half of the nodes from $A_i$ are removed.
- These have end points in $M_i$.
- $C_i$ is a vertex cover of $G_i$.
- Then every edge has at least one end point in $C_i$.
- At least $1/4$ of the edges in $A_i$ are contained in $C_i$.
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^{i-1}|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.
- Case 2: $|A_i| > 4|C_i|/3$.
  - Half of the nodes from $A_i$ are removed.
  - These have end points in $M_i$.
  - $C_i$ is a vertex cover of $G_i$.
  - Then every edge has at least one end point in $C_i$.
  - At least $1/4$ of the edges in $A_i$ are contained in $C_i$.
  - $C_i/G_{i+1}$ is a vertex cover of $G_{i+1}$.
Outer loop (Proof)

- We show using induction that $G_i$ contains a vertex cover $C_i$ with $|C_i| \leq (2/3)^i - 1|V|$.
- We will show that $|C_{i+1}| \leq 2|C_i|/3$.
- Basis: $i = 1$: Choose $C_1 = V$.
- Case 1: $|A_i| \leq 4|C_i|/3$.
  - In phase $i$ half of the nodes are removed from $A_i$.
  - $A_i/G_{i+1}$ is a vertex cover from $G_{i+1}$.
  - $|A_i|/2 \leq (4|C_i|/3)/2 = 2|C_i|/3$.
- Case 2: $|A_i| > 4|C_i|/3$.
  - Half of the nodes from $A_i$ are removed.
  - These have end points in $M_i$.
  - $C_i$ is a vertex cover of $G_i$.
  - Then every edge has at least one end point in $C_i$.
  - At least $1/4$ of the edges in $A_i$ are contained in $C_i$.
  - $C_i/G_{i+1}$ is a vertex cover of $G_{i+1}$.
  - $|C_i/G_{i+1}| \leq |C_i| - |A_i|/4 \leq |C_i| - (4|C_i|/3)/4 = 2|C_i|/3$. 

$|M_i| \geq |E_k|/4$

$|A_i/G_{i+1}| \leq |A_i|/2$
Summary

Theorem:

A maximal vertex cover can be computed in time $O(\log^4 n)$ using $O(n + m)$ processors.

$|M_i| \geq \frac{|E_k|}{4}$

$\frac{|A_i|}{G_{i+1}} \leq \frac{|A_i|}{2}$
Summary

Theorem:

A maximal vertex cover can be computed in time $O(\log^4 n)$ using $O(n + m)$ processors.

Proof:

- Outer loop: $O(\log n)$
Summary

Theorem:

A maximal vertex cover can be computed in time $O(\log^4 n)$ using $O(n + m)$ processors.

Proof:

- Outer loop: $O(\log n)$
- Inner loop: $O(\log n)$
A maximal vertex cover can be computed in time $O(\log^4 n)$ using $O(n + m)$ processors.

Proof:

- Outer loop: $O(\log n)$
- Inner loop: $O(\log n)$
- Running time of *DegreeSplit*: $O(\log^2 n)$. 
Bipartite graphs

A graph $G = (A, B, E)$ with $E \subseteq A \times B$ is called bipartite graph.
**Bipartite graphs**

A graph $G = (A, B, E)$ with $E \subseteq A \times B$ is called bipartite graph.

**Directed line graph**

Let $G = (V, E)$ be a directed graph, then $G^2 = (E, F)$ with $F = \{((a, b), (b, c)) \mid (a, b), (b, c) \in E\}$ is the line graph of $G$. 

**Edge coloring of bipartite graphs**

Compute a $k$-coloring of $G^2$. 

---

**3:38 Bipartite graphs 2/5**

Walter Unger 29.11.2016 20:11 WS2016/17
Edge coloring of bipartite graphs

**Bipartite graphs**

A graph $G = (A, B, E)$ with $E \subset A \times B$ is called bipartite graph.

**directed line graph**

Let $G = (V, E)$ be a directed graph, then $G^2 = (E, F)$ with $F = \{( (a, b), (b, c) ) \mid (a, b), (b, c) \in E \}$ is the line graph of $G$.

**undirected line graph**

Let $G = (V, E)$ be an undirected graph, then $G^2 = (E, F)$ with $F = \{ \{a, b\}, \{b, c\}\} \mid \{a, b\}, \{b, c\} \in E \}$ is the line graph of $G$. 
Bipartite graphs
A graph \( G = (A, B, E) \) with \( E \subset A \times B \) is called bipartite graph.

directed line graph
Let \( G = (V, E) \) be a directed graph, then \( G^2 = (E, F) \) with 
\[
F = \{((a, b), (b, c)) \mid (a, b), (b, c) \in E\}
\] is the line graph of \( G \).

undirected line graph
Let \( G = (V, E) \) be an undirected graph, then \( G^2 = (E, F) \) with 
\[
F = \{\{\{a, b\}, \{b, c\}\} \mid \{a, b\}, \{b, c\} \in E\}
\] is the line graph of \( G \).

Edge coloring
- Let \( G = (V, E) \) be an undirected graph and \( k \in \mathbb{N} \).
Edge coloring of bipartite graphs

Bipartite graphs

A graph $G = (A, B, E)$ with $E \subseteq A \times B$ is called bipartite graph.

directed line graph

Let $G = (V, E)$ be a directed graph, then $G^2 = (E, F)$ with $F = \{((a, b), (b, c)) \mid (a, b), (b, c) \in E\}$ is the line graph of $G$.

undirected line graph

Let $G = (V, E)$ be an undirected graph, then $G^2 = (E, F)$ with $F = \{\{\{a, b\}, \{b, c\}\} \mid \{a, b\}, \{b, c\} \in E\}$ is the line graph of $G$.

Edge coloring

- Let $G = (V, E)$ be an undirected graph and $k \in \mathbb{N}$.
- Compute $\exists k$-coloring of $G^2$. 
Introduction

- Let \( G = (V, E) \) be an undirected graph.
Introduction

- Let $G = (V, E)$ be an undirected graph.
- It is NP-complete to find a $\Delta(G)$ edge coloring.
Let $G = (V, E)$ be an undirected graph.

It is NP-complete to find a $\Delta(G)$ edge coloring.

There is always a $\Delta(G) + 1$ edge coloring.
Introduction

- Let $G = (V, E)$ be an undirected graph.
- It is NP-complete to find a $\Delta(G)$ edge coloring.
- There is always a $\Delta(G) + 1$ edge coloring.
- A bipartite graph $G$ is $\Delta(G)$ edge colorable.
Introduction

- Let $G = (V, E)$ be an undirected graph.
- It is NP-complete to find a $\Delta(G)$ edge coloring.
- There is always a $\Delta(G) + 1$ edge coloring.
- A bipartite graph $G$ is $\Delta(G)$ edge colorable.
- Or: A bipartite graph $G$ can be covered with $\Delta(G)$ matchings.
Introduction

- Let $G = (V, E)$ be an undirected graph.
- It is NP-complete to find a $\Delta(G)$ edge coloring.
- There is always a $\Delta(G) + 1$ edge coloring.
- A bipartite graph $G$ is $\Delta(G)$ edge colorable.
- Or: A bipartite graph $G$ can be covered with $\Delta(G)$ matchings.
- Here: Parallel edge coloring of a bipartite graph.
Introduction

- Let $G = (V, E)$ be an undirected graph.
- It is NP-complete to find a $\Delta(G)$ edge coloring.
- There is always a $\Delta(G) + 1$ edge coloring.
- A bipartite graph $G$ is $\Delta(G)$ edge colorable.
- Or: A bipartite graph $G$ can be covered with $\Delta(G)$ matchings.
- Here: Parallel edge coloring of a bipartite graph.
- 1.Step: $\Delta(G) = 2^k$ for some $k \in \mathbb{N}$. 
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
- Color edges alternating with 0 and 1.
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
- Color edges alternating with 0 and 1.
- This computes a partition of $G$ in $G_0$ and $G_1$ with $\Delta(G_0) = \Delta(G_1) = 2^{k-1}$. 
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
- Color edges alternating with 0 and 1.
- This computes a partition of $G$ in $G_0$ and $G_1$ with $\Delta(G_0) = \Delta(G_1) = 2^{k-1}$.
- All steps can be done in time $O(\log n)$ with $O(m)$ processors.
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
- Color edges alternating with 0 and 1.
- This computes a partition of $G$ in $G_0$ and $G_1$ with $\Delta(G_0) = \Delta(G_1) = 2^{k-1}$.
- All steps can be done in time $O(\log n)$ with $O(m)$ processors.
- Continue recursively.
Method for $\Delta(G) = 2^k$

- Idea: Cover the edges of $G$ with cycles and paths.
- Color edges alternating with 0 and 1.
- This computes a partition of $G$ in $G_0$ and $G_1$ with $\Delta(G_0) = \Delta(G_1) = 2^{k-1}$.
- All steps can be done in time $O(\log n)$ with $O(m)$ processors.
- Continue recursively.
- Total running time: $O(\log^2 n)$ with $O(m)$ processors.
Example
Example

[Graph representation of a bipartite graph with colored edges]
Example
Example
Example
Example
Example
Example
Example
Example
Example
Example
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^k$. 
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^{k'}$.
- Allow double coloring of edges, i.e. $(i,j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$. 
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the subgraph $G'$ with $\Delta(G') = 2^{k'}$.
- Allow double coloring of edges, i.e. $(i, j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$.
- Within each step it holds:
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^{k'}$.
- Allow double coloring of edges, i.e. $(i, j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$.
- Within each step it holds:
  - There are correctly colored edges and double colored edges.
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^{k'}$.
- Allow double coloring of edges, i.e. $(i, j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$.
- Within each step it holds:
  - There are correctly colored edges and double colored edges.
  - The set of colors $S$ is chosen such that the number of double colored edges is as big as possible,
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^k'$.

- Allow double coloring of edges, i.e. $(i, j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$.

- Within each step it holds:
  - There are correctly colored edges and double colored edges.
  - The set of colors $S$ is chosen such that the number of double colored edges is as big as possible,
  - These edges become colored correctly.
Method for $\Delta(G) < 2^k$ (Idea)

- Color as many edges as possible in the sub graph $G'$ with $\Delta(G') = 2^{k'}$.
- Allow double coloring of edges, i.e. $(i, j)$ is colored $\alpha$ at $i$ and $\beta$ at $j$.
- Within each step it holds:
  - There are correctly colored edges and double colored edges.
  - The set of colors $S$ is chosen such that the number of double colored edges is as big as possible.
  - These edges become colored correctly.
  - This happens in the extended sub graph with $\Delta(G') = 2^{k'}$. 
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$. 

Total running time: $O(\log 3 n)$ with $O(m)$ processors.
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \cdots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$. 

\[ \text{Total running time: } O(\log_3 n) \text{ with } O(m) \text{ processors.} \]
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'} + 1$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \cdots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
Method for \( \Delta(G) < 2^k \) (Idea)

- Let \( k' : 2^{k'} < \Delta(G) < 2^{k'+1} \), \( C = \emptyset \) and \( U = E \).
- Partition \( F = \{0, 1, 2, \ldots, \Delta(G) - 1\} \) into four sets of almost the same size \( S_1, S_2, S_3, S_4 \).
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from \( U \).
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$. 

Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
  - Let $U'$ be those edges.
Method for $\Delta(G) < 2^k$ (Idea)

1. Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.

2. Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.

3. Repeat until all edges are colored correctly:
   - Choose double coloring of the edges from $U$.
   - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
   - Let $U'$ be those edges.
   - It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$. 
Method for $\Delta(G) < 2^k$ (Idea)

Let $k': 2^k' < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.

Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.

Repeat until all edges are colored correctly:

- Choose double coloring of the edges from $U$.
- Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
- Let $U'$ be those edges.
- It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$.
- Let $H$ be those edges that only use colors from $S_i \cup S_j$. 
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k'$: $2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
  - Let $U'$ be those edges.
  - It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$.
  - Let $H$ be those edges that only use colors from $S_i \cup S_j$.
  - Let $G' = (V, H)$, extend $G'$ such that $\Delta(G') = 2^{k'}$. 
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
  - Let $U'$ be those edges.
  - It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$.
  - Let $H$ be those edges that only use colors from $S_i \cup S_j$.
  - Let $G' = (V, H)$, extend $G'$ such that $\Delta(G') = 2^{k'}$.
  - Color $G'$ using the method from above.
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k': 2^{k'} < \Delta(G) < 2^{k'} + 1$, $C = \emptyset$ and $U = E$.

- Partition $F = \{0, 1, 2, \ldots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.

- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
  - Let $U'$ be those edges.
  - It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$.
  - Let $H$ be those edges that only use colors from $S_i \cup S_j$.
  - Let $G' = (V, H)$, extend $G'$ such that $\Delta(G') = 2^{k'}$.
  - Color $G'$ using the method from above.
  - Set $C = C \cup H$, these are the correctly colored edges.
Method for $\Delta(G) < 2^k$ (Idea)

- Let $k' : 2^k' < \Delta(G) < 2^{k'+1}$, $C = \emptyset$ and $U = E$.
- Partition $F = \{0, 1, 2, \cdots, \Delta(G) - 1\}$ into four sets of almost the same size $S_1, S_2, S_3, S_4$.
- Repeat until all edges are colored correctly:
  - Choose double coloring of the edges from $U$.
  - Chose $i, j$ with: As many edges as possible from $U$ are colored with only $S_i \cup S_j$.
  - Let $U'$ be those edges.
  - It holds: $|U'| \geq |U|/6$ and $U' \leq 2^{k'}$.
  - Let $H$ be those edges that only use colors from $S_i \cup S_j$.
  - Let $G' = (V, H)$, extend $G'$ such that $\Delta(G') = 2^{k'}$.
  - Color $G'$ using the method from above.
  - Set $C = C \cup H$, these are the correctly colored edges.

- Total running time: $O(\log^3 n)$ with $O(m)$ processors.
Example (1. round)
Example (1. round)
Example (1. round)
Example (1. round)
Example (2. round)
Example (2. round)
Example (2. round)
Example (2. round)
Example (2. round)
Example (3. round)
Example (3. round)
Example (3. round)
Example (3. round)
Example (3. round)
Example (result)
Theorem:

A bipartite graph $G$ with $\Delta(G) = 2^k$ can be edge colored with $\Delta(G)$ colors in time $O(\log^2 n)$ with $O(m)$ processors.

Proof: See above.

Theorem:

A bipartite graph $G$ can be edge colored with $\Delta(G)$ colors in time $O(\log^3 n)$ with $O(m)$ processors.

Proof: See above.
Results without proof

Lemma

Any graph \( G = (V, E) \) with maximal degree \( \Delta \) is \( \Delta + 1 \) colorable.
Results without proof

**Lemma**

Any graph $G = (V, E)$ with maximal degree $\Delta$ is $\Delta + 1$ colorable.

**Lemma**

Any graph $G = (V, E)$, which is not a clique nor a odd cycle is $\Delta$ colorable.
Results without proof

Lemma

Any graph $G = (V, E)$ with maximal degree $\Delta$ is $\Delta + 1$ colorable.

Lemma

Any graph $G = (V, E)$, which is not a clique nor a odd cycle is $\Delta$ colorable.

- Idea of distributed/parallel algorithm:
Results without proof

Lemma

Any graph $G = (V, E)$ with maximal degree $\Delta$ is $\Delta + 1$ colorable.

Lemma

Any graph $G = (V, E)$, which is not a clique nor an odd cycle is $\Delta$ colorable.

- Idea of distributed/parallel algorithm:
- Reduce recursively the colors.
Results without proof

**Lemma**

Any graph \( G = (V, E) \) with maximal degree \( \Delta \) is \( \Delta + 1 \) colorable.

**Lemma**

Any graph \( G = (V, E) \), which is not a clique nor a odd cycle is \( \Delta \) colorable.

- Idea of distributed/parallel algorithm:
  - Reduce recursively the colors.
  - Double the size of correctly colored sub-graphs.
Results without proof

**Lemma**

Any graph $G = (V, E)$ with maximal degree $\Delta$ is $\Delta + 1$ colorable.

**Lemma**

Any graph $G = (V, E)$, which is not a clique nor a odd cycle is $\Delta$ colorable.

- Idea of distributed/parallel algorithm:
  - Reduce recursively the colors.
  - Double the size of correctly colored sub-graphs.
  - Or use the idea for trees to bounded degree graphs.
Recall and Idea 1

Theorem:

A tree with $n$ nodes could be colored with $n$ processors in time $O(\log^*n)$ with at most 3 colors.
Recall and Idea 1

Theorem:

A tree with \( n \) nodes could be colored with \( n \) processors in time \( O(\log^* n) \) with at most 3 colors.

- Recall: choose minimal \( k \) with: \((c \gg k) \mod 2 \neq (c' \gg k) \mod 2\) and
Recall and Idea 1

Theorem:
A tree with \( n \) nodes could be colored with \( n \) processors in time \( O(\log^* n) \) with at most 3 colors.

- Recall: choose minimal \( k \) with: \( (c \gg k) \% 2 \neq (c' \gg k) \% 2 \) and
- set \( c = 2 \cdot k + ((c \gg k) \% 2) \).
Recall and Idea 1

Theorem:
A tree with \( n \) nodes could be colored with \( n \) processors in time \( O(\log^* n) \) with at most 3 colors.

- Recall: choose minimal \( k \) with: \( ((c \gg k) \% 2) \neq ((c' \gg k) \% 2) \) and
- set \( c = 2 \cdot k + ((c \gg k) \% 2) \).
- This did produce a 6-coloring on trees.
Recall and Idea 1

Theorem:
A tree with $n$ nodes could be colored with $n$ processors in time $O(\log^* n)$ with at most 3 colors.

- Recall: choose minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$ and
- set $c = 2 \cdot k + ((c \gg k) \% 2)$.
- This did produce a 6-coloring on trees.
- On a bounded degree graph use this idea on a vector of length $\Delta$. 
choose minimal \( k \) with: \( (c \gg k) \% 2 \neq (c' \gg k) \% 2 \) and set \( c = 2 \cdot k + (c \gg k) \% 2 \)

1. Let \( v_1, v_2, \ldots, v_d \) the \( d \leq \Delta \) neighbors of \( v \)
Algorithm 1

choose minimal $k$ with: $((c \gg k) \mod 2) \neq ((c' \gg k) \mod 2)$ and set $c = 2 \cdot k + ((c \gg k) \mod 2)$

1. Let $v_1, v_2, \ldots, v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, \ldots, c_d$ the colors $v_i$ and $c$ the color of $v$. 
Algorithm 1

choose minimal $k$ with: $((c \gg k) \mod 2) \neq ((c' \gg k) \mod 2)$ and set $c = 2 \cdot k + ((c \gg k) \mod 2)$

1. Let $v_1, v_2, ..., v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, ..., c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ ($1 \leq i \leq d$) do

   As before, the coloring stays valid. Like before, a $x$-bit coloring becomes a $\Delta(\log x + 1)$-bit coloring. Like before, we may reduce the colors to $\Delta + 1$ colors. For unbounded degree, the running time becomes $O(\log^* n + 2\Delta)$. 

Algorithm 1

choose minimal $k$ with: $((c \gg k) \mod 2) \neq ((c' \gg k) \mod 2)$ and set $c = 2 \cdot k + ((c \gg k) \mod 2)$

1. Let $v_1, v_2, \ldots, v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, \ldots, c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ ($1 \leq i \leq d$) do
   1. choose minimal $k_i$ with: $((c \gg k_i) \mod 2) \neq ((c_i \gg k_i) \mod 2)$ and
Algorithm 1

choose minimal $k$ with: $((c \gg k) \mod 2) \neq ((c' \gg k) \mod 2)$ and set $c = 2 \cdot k + ((c \gg k) \mod 2)$

1. Let $v_1, v_2, \ldots, v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, \ldots, c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ ($1 \leq i \leq d$) do
   1. choose minimal $k_i$ with: $((c \gg k_i) \mod 2) \neq ((c_i \gg k_i) \mod 2)$ and
   2. set $b_i = 2 \cdot k_i + ((c \gg k_i) \mod 2)$. 

As before, the coloring stays valid.

Like before, a $x$-bit coloring becomes a $\Delta(\log x + 1)$-bit coloring.

Like before, we may reduce the colors to $\Delta + 1$ colors.

For unbounded degree the running time becomes: $O(\log^* n + 2 \Delta)$. 

Algorithm 1

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

1. Let $v_1, v_2, ..., v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, ..., c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ ($1 \leq i \leq d$) do
   1. choose minimal $k_i$ with: $((c \gg k_i)\%2) \neq ((c_i \gg k_i)\%2)$ and
   2. set $b_i = 2 \cdot k_i + ((c \gg k_i)\%2)$.
4. Choose new color for $v$: $(b_1, b_2, ..., b_d)$.
Algorithm 1

choose minimal \( k \) with: \( ((c \gg k)\%2) \neq ((c' \gg k)\%2) \) and set \( c = 2 \cdot k + ((c \gg k)\%2) \)

1. Let \( v_1, v_2, \ldots, v_d \) the \( d \leq \Delta \) neighbors of \( v \)
2. Let \( c_1, c_2, \ldots, c_d \) the colors \( v_i \) and \( c \) the color of \( v \).
3. For each \( i \, (1 \leq i \leq d) \) do
   1. choose minimal \( k_i \) with: \( ((c \gg k_i)\%2) \neq ((c_i \gg k_i)\%2) \) and
   2. set \( b_i = 2 \cdot k_i + ((c \gg k_i)\%2) \).
4. Choose new color for \( v \): \( (b_1, b_2, \ldots, b_d) \).
Algorithm 1

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

1. Let $v_1, v_2, \ldots, v_d$ the $d \leq \Delta$ neighbors of $v$

2. Let $c_1, c_2, \ldots, c_d$ the colors $v_i$ and $c$ the color of $v$.

3. For each $i$ ($1 \leq i \leq d$) do
   1. choose minimal $k_i$ with: $((c \gg k_i)\%2) \neq ((c_i \gg k_i)\%2)$ and
   2. set $b_i = 2 \cdot k_i + ((c \gg k_i)\%2)$.

4. Choose new color for $v$: $(b_1, b_2, \ldots, b_d)$.

- As before, the coloring stays valid.
Algorithm 1

choose minimal \( k \) with: \( ((c \gg k)\%2) \neq ((c' \gg k)\%2) \) and set \( c = 2 \cdot k + ((c \gg k)\%2) \)

1. Let \( v_1, v_2, ..., v_d \) the \( d \leq \Delta \) neighbors of \( v \)
2. Let \( c_1, c_2, ..., c_d \) the colors \( v_i \) and \( c \) the color of \( v \).
3. For each \( i \) (\( 1 \leq i \leq d \)) do
   1. choose minimal \( k_i \) with: \( ((c \gg k_i)\%2) \neq ((c_i \gg k_i)\%2) \) and
   2. set \( b_i = 2 \cdot k_i + ((c \gg k_i)\%2) \).
4. Choose new color for \( v \): \( (b_1, b_2, ..., b_d) \).

- As before, the coloring stays valid.
- Like before, a \( x \)-bit coloring becomes a \( \Delta(\log x + 1) \)-bit coloring.
Algorithm 1

choose minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$ and set $c = 2 \cdot k + ((c \gg k) \% 2)$

1. Let $v_1, v_2, \ldots, v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, \ldots, c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ $(1 \leq i \leq d)$ do
   1. choose minimal $k_i$ with: $((c \gg k_i) \% 2) \neq ((c_i \gg k_i) \% 2)$ and
   2. set $b_i = 2 \cdot k_i + ((c \gg k_i) \% 2)$.
4. Choose new color for $v$: $(b_1, b_2, \ldots, b_d)$.

- As before, the coloring stays valid.
- Like before, a $x$-bit coloring becomes a $\Delta(\log x + 1)$-bit coloring.
- Like before, we may reduce the colors to $\Delta + 1$ colors.
Algorithm 1

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

1. Let $v_1, v_2, ..., v_d$ the $d \leq \Delta$ neighbors of $v$
2. Let $c_1, c_2, ..., c_d$ the colors $v_i$ and $c$ the color of $v$.
3. For each $i$ (1 $\leq i \leq d$) do
   1. choose minimal $k_i$ with: $((c \gg k_i)\%2) \neq ((c_i \gg k_i)\%2)$ and
   2. set $b_i = 2 \cdot k_i + ((c \gg k_i)\%2)$.
4. Choose new color for $v$: $(b_1, b_2, ..., b_d)$.

- As before, the coloring stays valid.
- Like before, a $x$-bit coloring becomes a $\Delta(\log x + 1)$-bit coloring.
- Like before, we may reduce the colors to $\Delta + 1$ colors.
- For unbounded degree the running time becomes: $O(\log^* n + 2^\Delta)$. 
Theorem 1

A constant degree graph may be colored with $\Delta + 1$ colors in time $O(\log^* n)$ on a distributed system.

choose minimal $k$ with: $((c \gg k) \mod 2) \neq ((c' \gg k) \mod 2)$ and set $c = 2 \cdot k + ((c \gg k) \mod 2)$
Theorem

A constant degree graph may be colored with $\Delta + 1$ colors in time $O(\log^* n)$ on a distributed system.

Theorem

A constant degree graph may be colored with $\Delta + 1$ colors in time $O(\log^* n)$ on a parallel system using $n$ processors.
Notations and Idea 2

choose minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$ and set $c = 2 \cdot k + ((c \gg k) \% 2)$

- $x$ will be a binary string with up to $k$ bits.
choose minimal $k$ with: $((c \gg k) \% 2) \neq ((c' \gg k) \% 2)$ and set $c = 2 \cdot k + ((c \gg k) \% 2)$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{(a_1, a_2, \ldots a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$. 
Notations and Idea 2

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{(a_1, a_2, \ldots a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$.
- The procedure RecurseColor will color $U_x$ with $\Delta + 1$ colors.
Notations and Idea 2

choose minimal $k$ with: $(c \gg k) \% 2 \neq (c' \gg k) \% 2$ and set $c = 2 \cdot k + (c \gg k) \% 2$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{(a_1, a_2, ... a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$.
- The procedure RecurseColor will color $U_x$ with $\Delta + 1$ colors.
- Idea:
Notations and Idea 2

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{ (a_1, a_2, ... a_{k-|x|}, x) \mid a_i \in \{0, 1\} \}$.
- The procedure RecurseColor will color $U_x$ with $\Delta + 1$ colors.
- Idea:
  - Having colored $U_x$ with $\Delta + 1$ colors,
Notations and Idea 2

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{(a_1, a_2, ... a_{k-|x|}, x) | a_i \in \{0, 1\}\}$.
- The procedure RecurseColor will color $U_x$ with $\Delta + 1$ colors.
- Idea:
  - Having colored $U_x$ with $\Delta + 1$ colors,
  - Recolor $U_{1x}$ such that $U_{0x}$ and $U_{1x}$ are colored correctly.
Notations and Idea 2

choose minimal $k$ with: $((c \gg k)\%2) \neq ((c' \gg k)\%2)$ and set $c = 2 \cdot k + ((c \gg k)\%2)$

- $x$ will be a binary string with up to $k$ bits.
- Define $U_x = \{(a_1, a_2, ... a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$.
- The procedure RecurseColor will color $U_x$ with $\Delta + 1$ colors.
- Idea:
  - Having colored $U_x$ with $\Delta + 1$ colors,
  - Recolor $U_{1x}$ such that $U_{0x}$ and $U_{1x}$ are colored correctly.
  - This doubles the size of correctly colored sub-graphs.
Recursive Algorithm

RecurseColor($x$) (initial with $x = \varepsilon$):

1. Let $ID = (a_1, a_2, ..., a_k)$ be a vector of bits, which identify the node/prozessor $v$.

$$U_x = \{(a_1, a_2, ..., a_k-|x|, x) \mid a_i \in \{0, 1\}\}$$

Theorem

A graph of degree $\Delta$ may be colored with $\Delta + 1$ colors in time $O(\Delta \log n)$ on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with $x = \varepsilon$):

1. Let $ID = (a_1, a_2, ..., a_k)$ be a vector of bits, which identify the node/prozessor $v$.
2. Set $l = |x|$.

Let $ID = (a_1, a_2, ..., a_k)$ be a vector of bits, which identify the node/prozessor $v$.

Set $l = |x|$.

Theorem

A graph of degree $\Delta$ may be colored with $\Delta + 1$ colors in time $O(\Delta \log n)$ on a distributed/parallel system.
Recursive Algorithm

RecureColor(x) (initial with x = 0):  

1. Let ID = (a₁, a₂, ..., aₖ) be a vector of bits, which identify the node/prozessor v.  
2. Set l = |x|.  
3. If l = k then set c(v) = 1 and return.

\[ U_x = \{(a_1, a_2, ..., a_k - |x|, x) \mid a_i \in \{0, 1\}\} \]

Theorem  

A graph of degree Δ may be colored with Δ + 1 colors in time \(O(\Delta \log n)\) on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with x = ε):

1. Let ID = (a₁, a₂, ..., aₖ) be a vector of bits, which identify the node/prozessor v.
2. Set l = |x|.
3. If l = k then set c(v) = 1 and return.
4. Set b = aₖ−l.

\[ U_x = \{(a_1, a_2, ..., a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

Theorem

A graph of degree \( \Delta \) may be colored with \( \Delta + 1 \) colors in time \( O(\Delta \log n) \) on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with $x = \varepsilon$):

1. Let $ID = (a_1, a_2, \ldots, a_k)$ be a vector of bits, which identify the node/prozessor $v$.
2. Set $l = |x|$.
3. If $l = k$ then set $c(v) = 1$ and return.
4. Set $b = a_{k-l}$.
5. Set $c(v) = \text{RecurseColor}(bx)$.

Theorem

A graph of degree $\Delta$ may be colored with $\Delta + 1$ colors in time $O(\Delta \log n)$ on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with x = ε):

1. Let ID = (a_1, a_2, ..., a_k) be a vector of bits, which identify the node/prozessor v.
2. Set l = |x|.
3. If l = k then set c(v) = 1 and return.
4. Set b = a_{k-1}.
5. Set c(v) = RecurseColor(bx).
6. If b = 0 then return.

\[ U_x = \{(a_1, a_2, ...a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

Theorem

A graph of degree $\Delta$ may be colored with $\Delta + 1$ colors in time $O(\Delta \log n)$ on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with x = ε):

1. Let ID = (a₁, a₂, ..., aₖ) be a vector of bits, which identify the node/prozessor v.
2. Set l = |x|.
3. If l = k then set c(v) = 1 and return.
4. Set b = aₖ−l.
5. Set c(v) = RecurseColor(bx).
6. If b = 0 then return.
7. For round i from 1 to Δ + 1 do

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

Theorem

A graph of degree Δ may be colored with Δ + 1 colors in time \(O(Δ \log n)\) on a distributed/parallel system.
Recursive Algorithm

RecurseColor(x) (initial with $x = \epsilon$):

1. Let $ID = (a_1, a_2, ..., a_k)$ be a vector of bits, which identify the node/prozessor $v$.
2. Set $l = |x|$.
3. If $l = k$ then set $c(v) = 1$ and return.
4. Set $b = a_k - l$.
5. Set $c(v) = \text{RecurseColor}(bx)$.
6. If $b = 0$ then return.
7. For round $i$ from 1 to $\Delta + 1$ do
   1. if $c(v) = i$ then $c(v) = \min\{1, 2, ..., \Delta + 1\} \cup \{v\} \cup \{c(a)\}$

**Theorem**

A graph of degree $\Delta$ may be colored with $\Delta + 1$ colors in time $O(\Delta \log n)$ on a distributed/parallel system.
Independent Set

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \not\in E \) is called independent set.
Independent Set

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \not\in E \) is called independent set.
- \( \alpha(G) = \max\{ |V'| ; V' \subset V \land \forall a, b \in V' : (a, b) \not\in E \} \).
Independent Set

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \notin E \) is called independent set.

- \( \alpha(G) = \max\{ |V'| : V' \subset V \land \forall a, b \in V' : (a, b) \notin E \} \).

- The problem of finding an independent set of size \( n/2 \) is NP-complete.
Independent Set

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \notin E \) is called independent set.
- \( \alpha(G) = \max \{ |V'| ; V' \subset V \land \forall a, b \in V' : (a, b) \notin E \} \).
- The problem of finding an independent set of size \( n/2 \) is NP-complete.
- A independent set \( I \) is call maximal iff there is no larger independent set containing \( I \).
Independent Set

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \notin E \) is called independent set.
- \( \alpha(G) = \max\{ |V'| ; \ V' \subset V \ \land \ \forall a, b \in V' : (a, b) \notin E \} \).
- The problem of finding an independent set of size \( n/2 \) is NP-complete.
- A independent set \( I \) is call maximal iff there is no larger independent set containing \( I \).
- This is called MIS.
Independent Set

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- \( V' \subset V \) with \( \forall a, b \in V' : (a, b) \notin E \) is called independent set.

- \( \alpha(G) = \max \{ |V'| \mid V' \subset V \land \forall a, b \in V' : (a, b) \notin E \} \).

- The problem of finding an independent set of size \( n/2 \) is NP-complete.

- A independent set \( I \) is call maximal iff there is no larger independent set containing \( I \).

- This is called MIS.

- Finding the lexicographical first MIS is P-complete.
Independent Set

- \( V' \subseteq V \) with \( \forall a, b \in V' : (a, b) \notin E \) is called independent set.
- \( \alpha(G) = \max \{ |V'| ; \ V' \subseteq V \land \forall a, b \in V' : (a, b) \notin E \} \).
- The problem of finding an independent set of size \( n/2 \) is \( \text{NP-complete} \).
- A independent set \( I \) is call maximal iff there is no larger independent set containing \( I \).
- This is called MIS.
- Finding the lexicographical first MIS is \( \text{P-complete} \).
- Coloring and independent set have some relationship.
Independent Set

$U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$

- $V' \subset V$ with $\forall a, b \in V' : (a, b) \notin E$ is called independent set.
- $\alpha(G) = \max\{ |V'| ; \ V' \subset V \land \forall a, b \in V' : (a, b) \notin E \}$.
- The problem of finding an independent set of size $n/2$ is NP-complete.
- A independent set $I$ is call maximal iff there is no larger independent set containing $I$.
- This is called MIS.
- Finding the lexicographical first MIS is P-complete.
- Coloring and independent set have some relationship.
- The nodes of one color form an independent set.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]

- Idea: use a coloring to compute a MIS:
Independent Set and Coloring

- Idea: use a coloring to compute a MIS:
  1. For all nodes set $b(v) = 0$.

$$U_k = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$$
Independent Set and Coloring

- Idea: use a coloring to compute a MIS:
  1. For all nodes set $b(v) = 0$.
  2. For all $i$ from 1 to $\chi(G)$ do

$U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) \mid a_i \in \{0, 1\}\}$
Independent Set and Coloring

- Idea: use a coloring to compute a MIS:
  1. For all nodes set $b(v) = 0$.
  2. For all $i$ from 1 to $\chi(G)$ do
     - if $b(v) = 0$ then set $b(v) = 1$. 

$$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$$
Idea: use a coloring to compute a MIS:

1. For all nodes set $b(v) = 0$.
2. For all $i$ from 1 to $\chi(G)$ do
   - if $b(v) = 0$ then set $b(v) = 1$.
   - if some neighbor of $v$ has $b(v) = 1$ then set $b(v) = -1$.

$U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) | a_i \in \{0, 1\}\}$
Independent Set and Coloring

- Idea: use a coloring to compute a MIS:
  1. For all nodes set $b(v) = 0$.
  2. For all $i$ from 1 to $\chi(G)$ do
     1. if $b(v) = 0$ then set $b(v) = 1$.
     2. if some neighbor of $v$ has $b(v) = 1$ then set $b(v) = -1$.

- This will produce in time is $O(\chi(G))$. 

$U_x = \{(a_1, a_2, ..., a_{|x|}, x) \mid a_i \in \{0, 1\}\}$
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

There is a deterministic \( O(\log^* n) \) time algorithm for MIS on cycles, trees and bounded degree graphs of \( n \) processors.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

There is a deterministic \( O(\log^* n) \) time algorithm for MIS on cycles, trees and bounded degree graphs of \( n \) processors.

**Theorem**

There is a deterministic \( O(\Delta \log n) \) time algorithm for MIS on any graph of \( n \) processors.
Independent Set and Coloring

Theorem

There is a deterministic $O(\log^* n)$ time algorithm for MIS on cycles, trees and bounded degree graphs of $n$ processors.

Theorem

There is a deterministic $O(\Delta \log n)$ time algorithm for MIS on any graph of $n$ processors.

Theorem

Any deterministic distributed algorithm needs at least $1/2(\log^* n - 1)$ rounds to color a cycle of length $n$ with 3 colors.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots a_{|x|-1}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

*There is a deterministic \(O(\log^* n)\) time algorithm for MIS on cycles, trees and bounded degree graphs of \(n\) processors.*

**Theorem**

*There is a deterministic \(O(\Delta \log n)\) time algorithm for MIS on any graph of \(n\) processors.*

**Theorem**

*Any deterministic distributed algorithm needs at least \(1/2(\log^* n − 1)\) rounds to color a cycle of length \(n\) with 3 colors.*

**Theorem**

*Any deterministic distributed MIS algorithm on a cycle of length \(n\) uses \(1/2(\log^* n − 3)\) rounds.*
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{k - |x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( \frac{1}{2}(\log^* n - 3) \) rounds.

- We have a lower bound of \( \frac{1}{2}(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
Independent Set and Coloring

\[ U_x = \{ (a_1, a_2, \ldots a_{k - |x|}, x) | a_i \in \{0, 1\} \} \]

**Theorem**

Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2 (\log^* n - 3) \) rounds.

- We have a lower bound of \( 1/2 (\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
Independent Set and Coloring

\[
U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}
\]

**Theorem**

*Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.*

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
- Each node which is in the MIS colors itself with color 1.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
- Each node which is in the MIS colors itself with color 1.
- Each node which is in the MIS sends a 2 to the neighbor to the right.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

*Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.*

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
- Each node which is in the MIS colors itself with color 1.
- Each node which is in the MIS sends a 2 to the neighbor to the right.
- Each node receiving a 2 colors itself with color 2.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
- Each node which is in the MIS colors itself with color 1.
- Each node which is in the MIS sends a 2 to the neighbor to the right.
- Each node receiving a 2 colors itself with color 2.
- Each node not receiving a 2 colors itself with color 3.
Independent Set and Coloring

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem**

*Any deterministic distributed MIS algorithm on a cycle of length \( n \) uses \( 1/2(\log^* n - 3) \) rounds.*

- We have a lower bound of \( 1/2(\log^* n - 1) \) for 3-coloring a cycle of length \( n \).
- We have to show, given a MIS we may color the cycle in just one more round.
- We may assume we have some cyclic order on the nodes.
- Each node which is in the MIS colors itself with color 1.
- Each node which is in the MIS sends a 2 to the neighbor to the right.
- Each node receiving a 2 colors itself with color 2.
- Each node not receiving a 2 colors itself with color 3.
- There are no non-colored nodes (see definition of MIS).
Planar graphs

\[ U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) \mid a_i \in \{0, 1\}\} \]

**Definition**

A graph \( G = (V, E) \) is called planar if there is an embedding into the plane without crossings.

- It holds for planar graphs that \( |E| \leq 3 \cdot |V| - 6. \)
Planar graphs

Definition

A graph $G = (V, E)$ is called planar if there is an embedding into the plane without crossings.

- It holds for planar graphs that $|E| \leq 3 \cdot |V| - 6$.
- $K_{3,3}$ and $K_5$ are not planar.
Planar graphs

Definition

A graph $G = (V, E)$ is called planar if there is an embedding into the plane without crossings.

- It holds for planar graphs that $|E| \leq 3 \cdot |V| - 6$.
- $K_{3,3}$ and $K_5$ are not planar.
- Planar graphs have nodes of degree $\leq 5$. 

$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$
Planar graphs

Definition

A graph $G = (V, E)$ is called planar if there is an embedding into the plane without crossings.

- It holds for planar graphs that $|E| \leq 3 \cdot |V| - 6$.
- $K_{3,3}$ and $K_5$ are not planar.
- Planar graphs have nodes of degree $\leq 5$.
- Planar graphs are 4 colorable.
Planar graphs

\[ U_x = \{(a_1, a_2, \ldots a_{k-\lvert x\rvert}, x) \mid a_i \in \{0, 1\}\} \]

**Definition**

A graph \( G = (V, E) \) is called planar if there is an embedding into the plane without crossings.

- It holds for planar graphs that \( \lvert E \rvert \leq 3 \cdot \lvert V \rvert - 6 \).
- \( K_{3,3} \) and \( K_5 \) are not planar.
- Planar graphs have nodes of degree \( \leq 5 \).
- Planar graphs are 4 colorable.
- A window is a closed region which is limited by a path.
Outer planar graphs

Definition
A graph $G = (V, E)$ is outerplanar if there is an embedding into the plane without crossings such that all nodes lie on the outer window.

- It holds for outerplanar graphs that $|E| \leq 2 \cdot |V| - 3$. 

$$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$$
Outer planar graphs

Definition

A graph $G = (V, E)$ is outerplanar if there is an embedding into the plane without crossings such that all nodes lie on the outer window.

- It holds for outerplanar graphs that $|E| \leq 2 \cdot |V| - 3$.
- $K_{2,3}$ and $K_4$ are outerplanar.
Outer planar graphs

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Definition**

A graph \( G = (V, E) \) is outerplanar if there is an embedding into the plane without crossings such that all nodes lie on the outer window.

- It holds for outerplanar graphs that \(|E| \leq 2 \cdot |V| - 3\).
- \( K_{2,3} \) and \( K_4 \) are outerplanar.
- **Outer planar graphs have nodes with degree \( \leq 2 \).**
**Outer planar graphs**

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Definition**

A graph \( G = (V, E) \) is outerplanar if there is an embedding into the plane without crossings such that all nodes lie on the outer window.

- It holds for outerplanar graphs that \(|E| \leq 2 \cdot |V| - 3\).
- \( K_{2,3} \) and \( K_4 \) are outerplanar.
- Outer planar graphs have nodes with degree \( \leq 2 \).
- Outer planar graphs are 3 colorable.
Outer planar graphs

A graph $G = (V, E)$ is outerplanar if there is an embedding into the plane without crossings such that all nodes lie on the outer window.

- It holds for outerplanar graphs that $|E| \leq 2 \cdot |V| - 3$.
- $K_{2,3}$ and $K_4$ are outerplanar.
- Outer planar graphs have nodes with degree $\leq 2$.
- Outer planar graphs are 3 colorable.
- The inner windows form a tree.
Overview of the Algorithm

Let $G$ be a connected outerplanar graph.
Overview of the Algorithm

- Let $G$ be a connected outerplanar graph.
- Compute the outer edges.

Let $G$ be a connected outerplanar graph.

$$U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$$
Overview of the Algorithm

- Let $G$ be a connected outerplanar graph.
- Compute the outer edges.
- Direct the outer edges such that they form a cycle.

\[ U_x = \{(a_1, a_2, \ldots a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]
Overview of the Algorithm

Let $G$ be a connected outerplanar graph.

- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
Overview of the Algorithm

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

Let \( G \) be a connected outerplanar graph.

- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
- Compute a directed cycle for every window.
Overview of the Algorithm

Let $G$ be a connected outerplanar graph.

- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
- Compute a directed cycle for every window.
- Color every window independently.

\[
U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}
\]
Overview of the Algorithm

- Let $G$ be a connected outerplanar graph.
- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
- Compute a directed cycle for every window.
- Color every window independently.
- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.

$$U_x = \{(a_1, a_2, …a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$$
Overview of the Algorithm

Let $G$ be a connected outerplanar graph.

- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
- Compute a directed cycle for every window.
- Color every window independently.
- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
- Combine the cycles into pairs of layers of bigger correctly colored objects.

$$U_x = \{(a_1, a_2, ..., a_k-|x|), x) \mid a_i \in \{0, 1\}\}$$
Overview of the Algorithm

Let $G$ be a connected outerplanar graph.

- Compute the outer edges.
- Direct the outer edges such that they form a cycle.
- Determine the location and orientation of the inner edges and double those to two directed edges.
- Compute a directed cycle for every window.
- Color every window independently.
- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
- Combine the cycles into pairs of layers of bigger correctly colored objects.
- Repeat the last step until the whole graph is colored correctly.

$$U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$$
Details of the algorithm.

- Compute the outer edges.

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, ..., a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \{v, w\} if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.
Details of the algorithm.

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2/\log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3/\log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.

\[
U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}
\]
Details of the algorithm.

Let \( U_x = \{(a_1, a_2, \ldots, a_k - |x|, x) \mid a_i \in \{0, 1\}\} \)

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
  - Sort the edges lexicographical in \( K_1, K_2, \ldots, K_{2.m} \).
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
  - Sort the edges lexicographical in \( K_1, K_2, \ldots, K_{2 \cdot m} \).
  - Successor of \( K_x = (i, j) \) is \( K_{2 \cdot j} = (r, s) \) if \( s \neq i \).
Details of the algorithm.

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
  - Sort the edges lexicographical in \( K_1, K_2, \ldots, K_{2^m} \).
  - Successor of \( K_x = (i, j) \) is \( K_{2^j} = (r, s) \) if \( s \neq i \).
  - Successor of \( K_x = (i, j) \) is \( K_{2^j+1} = (r, s) \) if \( s \neq i \).
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Compute the outer edges.
  - Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
  - A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
  - Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
  - Sort the edges lexicographical in \( K_1, K_2, \ldots, K_{2m} \).
  - Successor of \( K_x = (i, j) \) is \( K_{2j} = (r, s) \) if \( s \neq i \).
  - Successor of \( K_x = (i, j) \) is \( K_{2j+1} = (r, s) \) if \( s \neq i \).
  - Choose a cycle.
Details of the algorithm.

\[U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}\]

- Compute the outer edges.
  - Test for edge \(\{v, w\}\) if \(G \setminus \{v, w\}\) separates.
  - A test: \(O(\log^2 n)\) time using \(O(n^2 / \log^2 n)\) processors.
  - Total: \(O(\log^2 n)\) time with \(O(n^3 / \log^2 n)\) processors.

- Direct the outer edges such that they form a cycle.
  - Create for every outer edge two opposing directed edges.
  - Sort the edges lexicographical in \(K_1, K_2, \ldots, K_{2 \cdot m}\).
  - Successor of \(K_x = (i, j)\) is \(K_{2 \cdot j} = (r, s)\) if \(s \neq i\).
  - Successor of \(K_x = (i, j)\) is \(K_{2 \cdot j+1} = (r, s)\) if \(s \neq i\).
  - Choose a cycle.
  - Determine the position of every node on the cycle.
Details of the algorithm.

Compute the outer edges.

- Test for edge \( \{v, w\} \) if \( G \setminus \{v, w\} \) separates.
- A test: \( O(\log^2 n) \) time using \( O(n^2 / \log^2 n) \) processors.
- Total: \( O(\log^2 n) \) time with \( O(n^3 / \log^2 n) \) processors.

Direct the outer edges such that they form a cycle.

- Create for every outer edge two opposing directed edges.
- Sort the edges lexicographical in \( K_1, K_2, \cdots, K_{2m} \).
- Successor of \( K_x = (i, j) \) is \( K_{2j} = (r, s) \) if \( s \neq i \).
- Successor of \( K_x = (i, j) \) is \( K_{2j+1} = (r, s) \) if \( s \neq i \).
- Choose a cycle.
- Determine the position of every node on the cycle.
- Total running time: \( O(\log n) \) time with \( O(n) \) processors.

\[
U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}
\]
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0,1\}\} \]

- Determine the location and orientation of the inner node.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \cdots at the node a is given by the location of the nodes a_1, a_2, \cdots on the cycle.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \( \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \cdots \) at the node \( a \) is given by the location of the nodes \( a_1, a_2, \cdots \) on the cycle.
  - Total running time: \( O(\log n) \) time with \( O(n) \) processors.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \( \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \ldots \) at the node \( a \) is given by the location of the nodes \( a_1, a_2, \ldots \) on the cycle.
  - Total running time: \( O(\log n) \) time with \( O(n) \) processors.

- Create for every outer edge two opposing directed edges.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \( \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \ldots \) at the node \( a \) is given by the location of the nodes \( a_1, a_2, \ldots \) on the cycle.
  - Total running time: \( O(\log n) \) time with \( O(n) \) processors.

- Create for every outer edge two opposing directed edges.
- Determine the directed cycle in every window.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \cdots at the node \(a\) is given by the location of the nodes \(a_1, a_2, \cdots\) on the cycle.
  - Total running time: \(O(\log n)\) time with \(O(n)\) processors.

- Create for every outer edge two opposing directed edges.

- Determine the directed cycle in every window.
  - Compute new successors using the order of the edges at every node.
Details of the algorithm.

\[ U_x = \left\{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \right\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \( \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \cdots \) at the node \( a \) is given by the location of the nodes \( a_1, a_2, \cdots \) on the cycle.
  - Total running time: \( O(\log n) \) time with \( O(n) \) processors.

- Create for every outer edge two opposing directed edges.

- Determine the directed cycle in every window.
  - Compute new successors using the order of the edges at every node.
  - Compute new cycles and representatives.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Determine the location and orientation of the inner node.
  - Sort the inner edges \( \{a, a_1\}, \{a, a_2\}, \{a, a_3\}, \ldots \) at the node \( a \) is given by the location of the nodes \( a_1, a_2, \ldots \) on the cycle.
  - Total running time: \( O(\log n) \) time with \( O(n) \) processors.

- Create for every outer edge two opposing directed edges.

- Determine the directed cycle in every window.
  - Compute new successors using the order of the edges at every node.
  - Compute new cycles and representatives.
  - Total running time: \( O(\log n) \) with \( O(n) \) processors.
Details of the algorithm.

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]

- Color every window independently.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Color every window independently.
- Total running time: \( O(\log^* n) \) with \( O(n) \) processors.
Details of the algorithm.

- Color every window independently.
  - Total running time: $O(\log^* n)$ with $O(n)$ processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.

\[ U_x = \{ (a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\} \} \]
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_k - |x|), x) \mid a_i \in \{0, 1\}\]
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Color every window independently.
  - Total running time: \( O(\log^* n) \) with \( O(n) \) processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
Details of the algorithm.

- Color every window independently.
  - Total running time: $O(\log^* n)$ with $O(n)$ processors.
- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: $O(\log n)$ using $O(n)$ processors.

$$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$$
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots a_k, x) \mid a_i \in \{0, 1\}\} \]

- Color every window independently.
  - Total running time: \(O(\log^* n)\) with \(O(n)\) processors.
- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: \(O(\log n)\) using \(O(n)\) processors.
- Combine the cycles into pairs of layers of bigger correctly colored objects.
Details of the algorithm.

- Color every window independently.
  - Total running time: $O(\log^* n)$ with $O(n)$ processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: $O(\log n)$ using $O(n)$ processors.

- Combine the cycles into pairs of layers of bigger correctly colored objects.
  - The child cycle orients itself to the coloring of the parent cycle.
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Color every window independently.
  - Total running time: \(O(\log^* n)\) with \(O(n)\) processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: \(O(\log n)\) using \(O(n)\) processors.

- Combine the cycles into pairs of layers of bigger correctly colored objects.
  - The child cycle orients itself to the coloring of the parent cycle.
  - Total: \(O(1)\) time with \(O(n)\) processors.
Details of the algorithm.

- Color every window independently.
  - Total running time: \( O(\log^* n) \) with \( O(n) \) processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: \( O(\log n) \) using \( O(n) \) processors.

- Combine the cycles into pairs of layers of bigger correctly colored objects.
  - The child cycle orients itself to the coloring of the parent cycle.
  - Total: \( O(1) \) time with \( O(n) \) processors.

- Repeat the last step until the whole graph is colored correctly.

\[
U_x = \{(a_1, a_2, \ldots, a_{|x|-1}, x) \mid a_i \in \{0, 1\}\}
\]
Details of the algorithm.

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

- Color every window independently.
  - Total running time: \( O(\log^* n) \) with \( O(n) \) processors.

- Determine the tree structure of the windows i.e. every cycle corresponds to nodes in the tree.
  - Using the inner edges the neighborhood can be read directly.
  - The depth of the nodes can be computed using the ranking in the list.
  - Total running time: \( O(\log n) \) using \( O(n) \) processors.

- Combine the cycles into pairs of layers of bigger correctly colored objects.
  - The child cycle orients itself to the coloring of the parent cycle.
  - Total: \( O(1) \) time with \( O(n) \) processors.

- Repeat the last step until the whole graph is colored correctly.
  - Total: \( O(\log n) \) time with \( O(n) \) processors.
Facts

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**

A two-connected outerplanar graph can be colored with three colors using time \(O(\log^2 n)\) and \(O(n^3/\log^2 n)\) processors.

**Proof:** See above.
Facts

\[ U_x = \{(a_1, a_2, \ldots, a_k, x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**

A two-connected outerplanar graph can be colored with three colors using time \( O(\log^2 n) \) and \( O(n^3 / \log^2 n) \) processors.

**Proof:** See above.

**Theorem:**

An outerplanar graph can be colored with three colors using time \( O(\log^2 n) \) and \( O(n^3 / \log^2 n) \) processors.

**Proof:** Use similarly the tree structure of the two connected components.
Facts

$U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\}$

**Theorem:**

A two-connected outerplanar graph can be colored with three colors using time $O(\log^2 n)$ and $O(n^3 / \log^2 n)$ processors.

**Proof:** See above.

**Theorem:**

An outerplanar graph can be colored with three colors using time $O(\log^2 n)$ and $O(n^3 / \log^2 n)$ processors.

**Proof:** Use similarly the tree structure of the two connected components.

**Theorem:**

A planar graph can be colored with six colors in time $O(\log^2 n)$ with $O(n)$ processors.

**Proof:** See exercise.
Results without proof

Theorem:
The edges of an outerplanar graph $G$ with $\Delta(G) \leq 3$ and known embedding in the plane can be colored using three colors in time $O(\log^2 n)$ with $O(n^2)$ processors.

Idea if the proof: Similar procedure then above.
Results without proof

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**

The edges of an outerplanar graph \( G \) with \( \Delta(G) \leq 3 \) and known embedding in the plane can be colored using three colors in time \( O(\log^2 n) \) with \( O(n^2) \) processors.

Idea if the proof: Similar procedure then above.

**Theorem:**

The edges of an outerplanar graph \( G \) with known embedding in the plane can be colored with three colors in time \( O(\log^3 n) \) with \( O(n^2) \) colors.

Proof: See literature.
Simulations

Theorem:

A program $A$ for a CREW PRAM with $P_A(n)$ processors and running time $T_A(n)$ can be simulated with an EREW PRAM with $P_A(n)^2$ processors in time $O(T_A(n) \log n)$.

$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$
Simulations

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**
A program \( A \) for a CREW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an EREW PRAM with \( P_A(n)^2 \) processors in time \( O(T_A(n) \log n) \).

**Theorem:**
A program \( A \) for a CRCW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an CREW PRAM with \( P_A(n)^2 \) processors in time \( O(T_A(n) \log n) \).
Simulations

\[ U_x = \{(a_1, a_2, \ldots, a_{k-|x|}, x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**
A program \( A \) for a CREW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an EREW PRAM with \( P_A(n)^2 \) processors in time \( O(T_A(n) \log n) \).

**Theorem:**
A program \( A \) for a CRCW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an CREW PRAM with \( P_A(n)^2 \) processors in time \( O(T_A(n) \log n) \).

**Theorem:**
A program \( A \) for a CRCW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an EREW PRAM with \( P_A(n)^2 \) processors in time \( O(T_A(n) \log n) \).
Simulations II

Theorem:

A program $A$ for a CREW PRAM with $P_A(n)$ processors and running time $T_A(n)$ can be simulated with an EREW PRAM with $P_A(n)$ processors in time $O(T_A(n) \log n)$.
Theorem:
A program $A$ for a CREW PRAM with $P_A(n)$ processors and running time $T_A(n)$ can be simulated with an EREW PRAM with $P_A(n)$ processors in time $O(T_A(n) \log n)$.

Theorem:
A program $A$ for a CRCW PRAM with $P_A(n)$ processors and running time $T_A(n)$ can be simulated with an CREW PRAM with $P_A(n)$ processors in time $O(T_A(n) \log n)$.

$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$
Simulations II

\[ U_x = \{(a_1, a_2, \ldots, a_k, x), x) \mid a_i \in \{0, 1\}\} \]

**Theorem:**
A program \( A \) for a CREW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an EREW PRAM with \( P_A(n) \) processors in time \( O(T_A(n) \log n) \).

**Theorem:**
A program \( A \) for a CRCW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an CREW PRAM with \( P_A(n) \) processors in time \( O(T_A(n) \log n) \).

**Theorem:**
A program \( A \) for a CRCW PRAM with \( P_A(n) \) processors and running time \( T_A(n) \) can be simulated with an EREW PRAM with \( P_A(n) \) processors in time \( O(T_A(n) \log n) \).
$U_x = \{(a_1, a_2, \ldots, a_{|x|}, x) \mid a_i \in \{0, 1\}\}$

Literature: