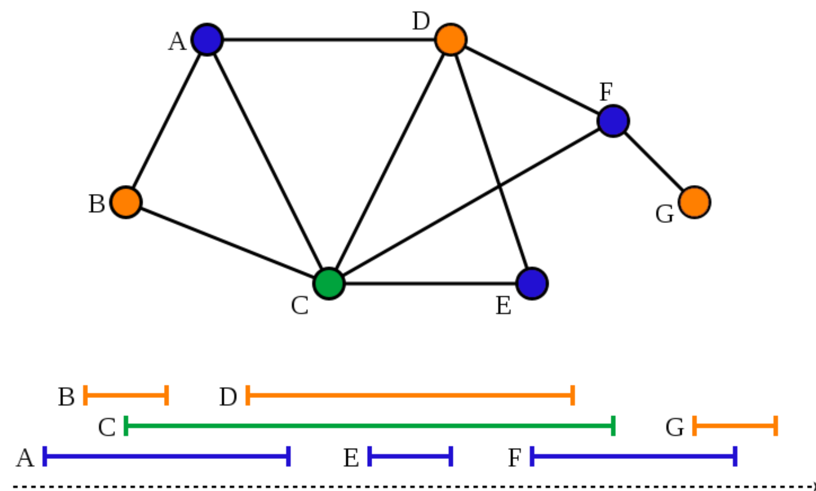
MODEL CHECKING PARALLEL FIRST FIT GRAPH COLORING IN JAVA

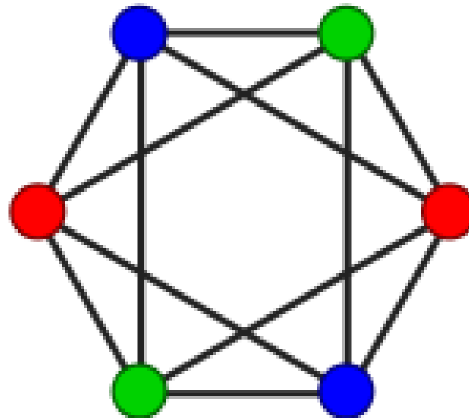
CSE 6490A Winter 2011

Loutfouz Zaman



Vertex coloring

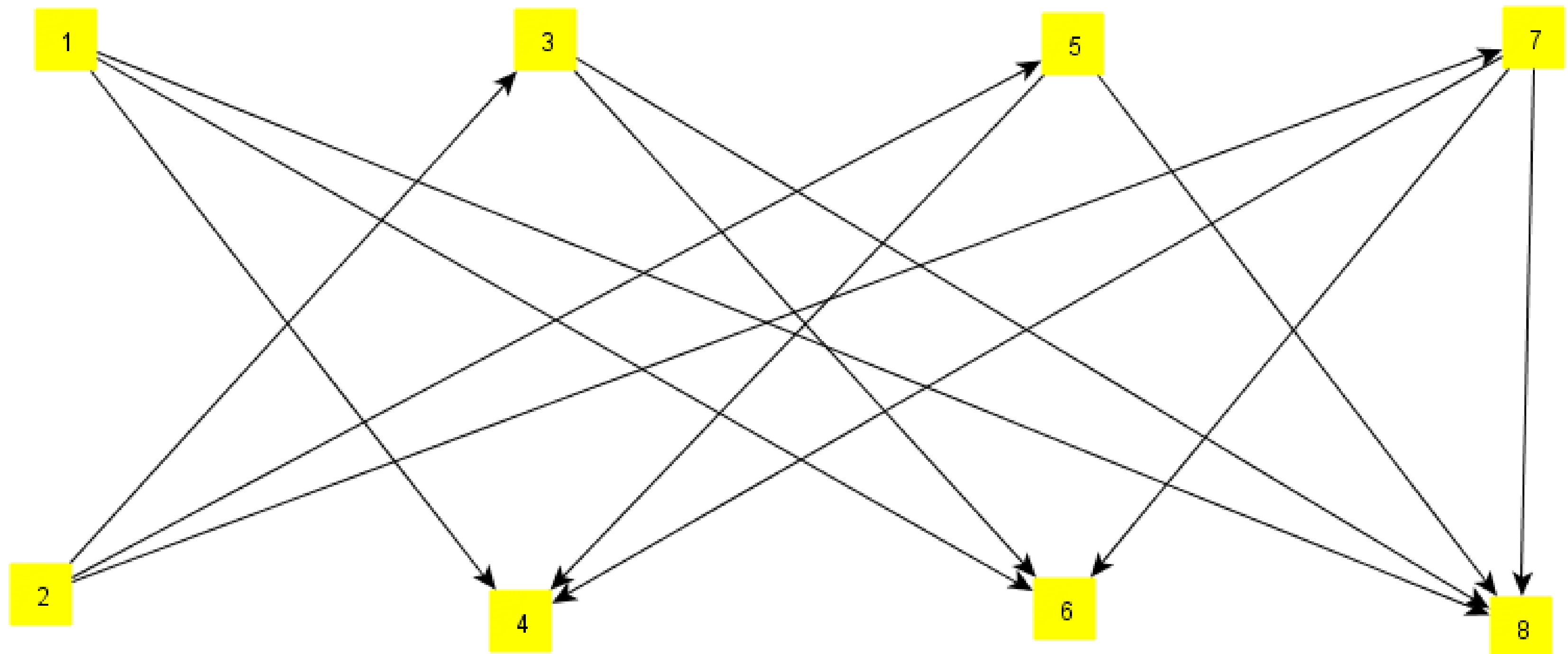
- Assignment of *"colors"* to vertices in a way so that no two adjacent vertices share the same color
- First-Fit is the simplest algorithm
  - works by assigning the smallest possible integer as color to the current vertex of the graph



Sequential FF

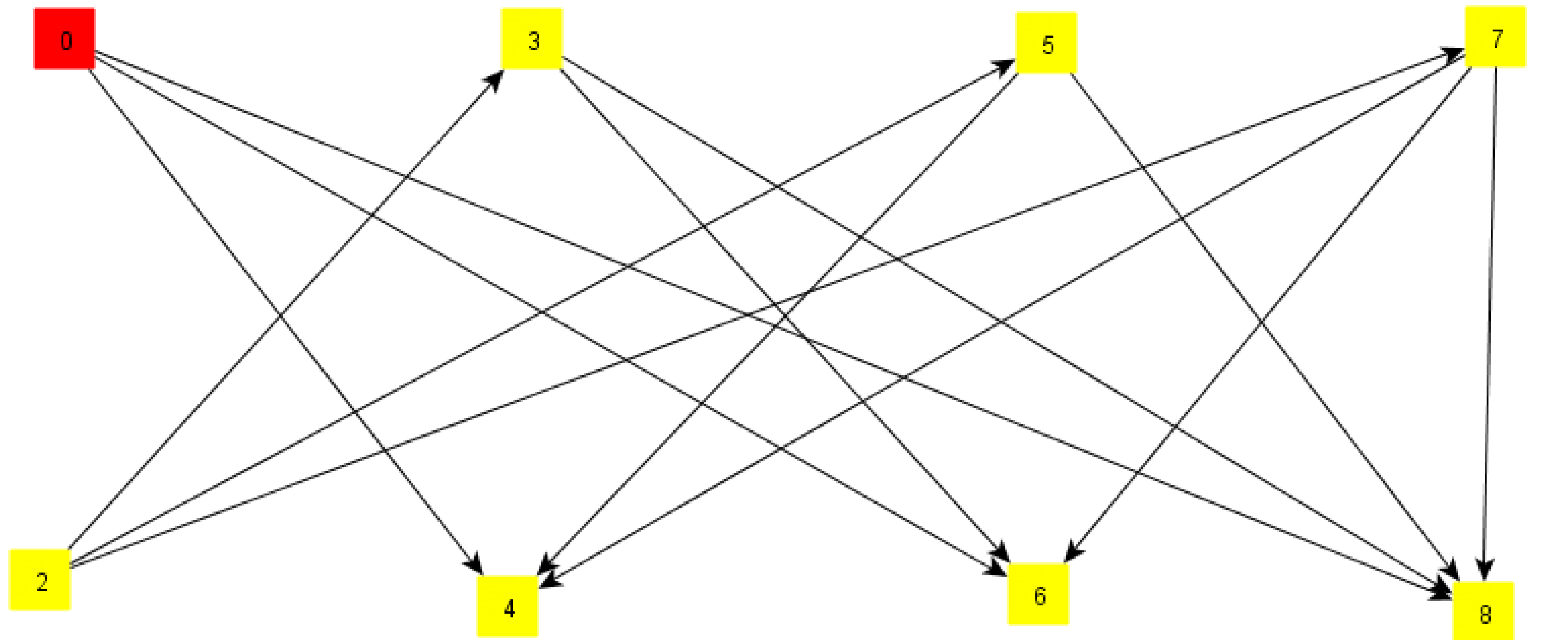
- Umland (1998) demonstrates a 2-step sequential FF algorithm:
  - **(1)** $\boldsymbol{Build(L_i, v_j)}$: Determine a list L_i of all possible colors for v_i, i.e. exclude colors already used by vertices $v_j, j < i$ adjacent to v_i
    - L_i -- a boolean array (possibility list of v_i) with property:
      - $L_i[k] = false \leftrightarrow \exists v_j$ such that $j < i, (v_i, v_j) \in E$ and $f(v_j) = k$
  - **(2)** $\boldsymbol{Color(L_i, v_i)}$: Determine the smallest of all possible colors for v_i, i.e. look for the smallest entry in L_i where $L_i[k] = true$ and assign color k to v_i

Sequential FF E.g. Step 0



Sequential FF E.g. Step 1

$L_1 = \{t, t, t, t\}$,k=0



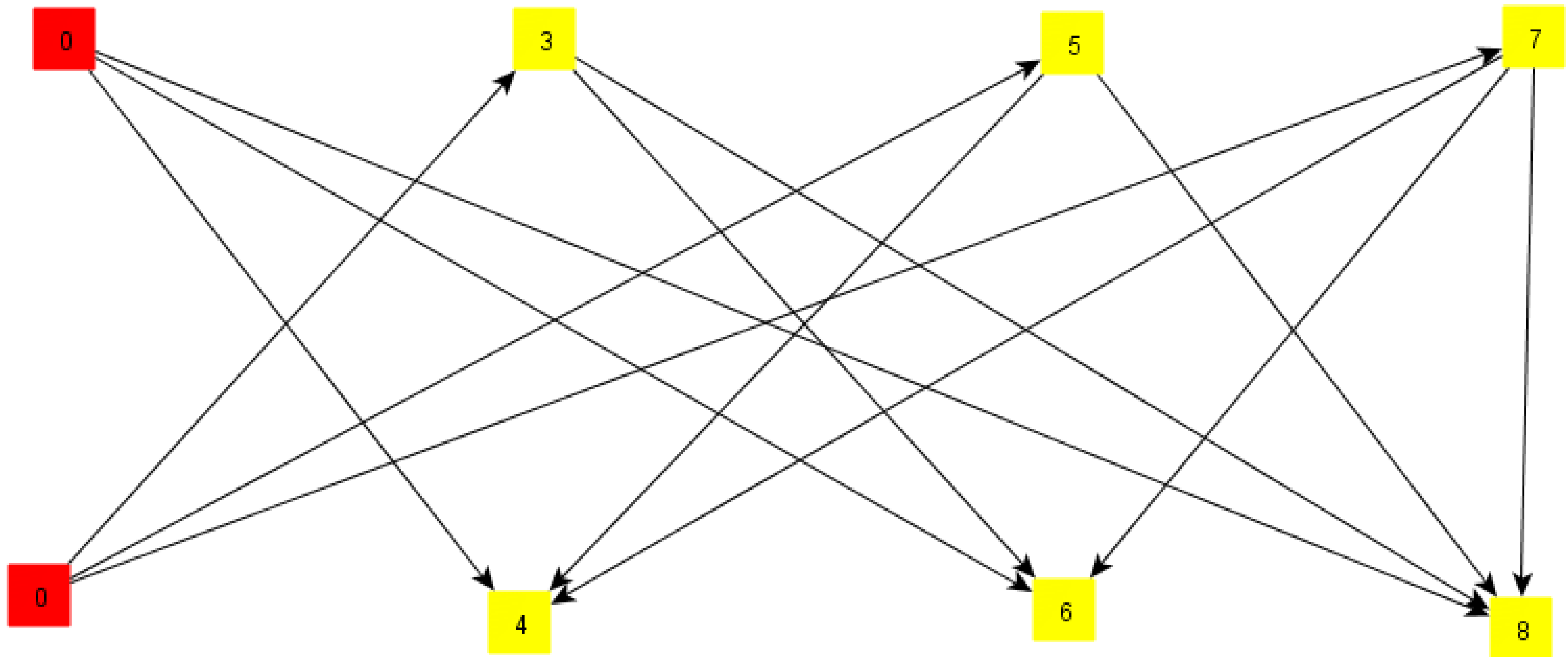
$L_4 = \{f, t, t, t\}$

$L_6 = \{f, t, t, t\}$

$L_6 = \{f, t, t, t\}$

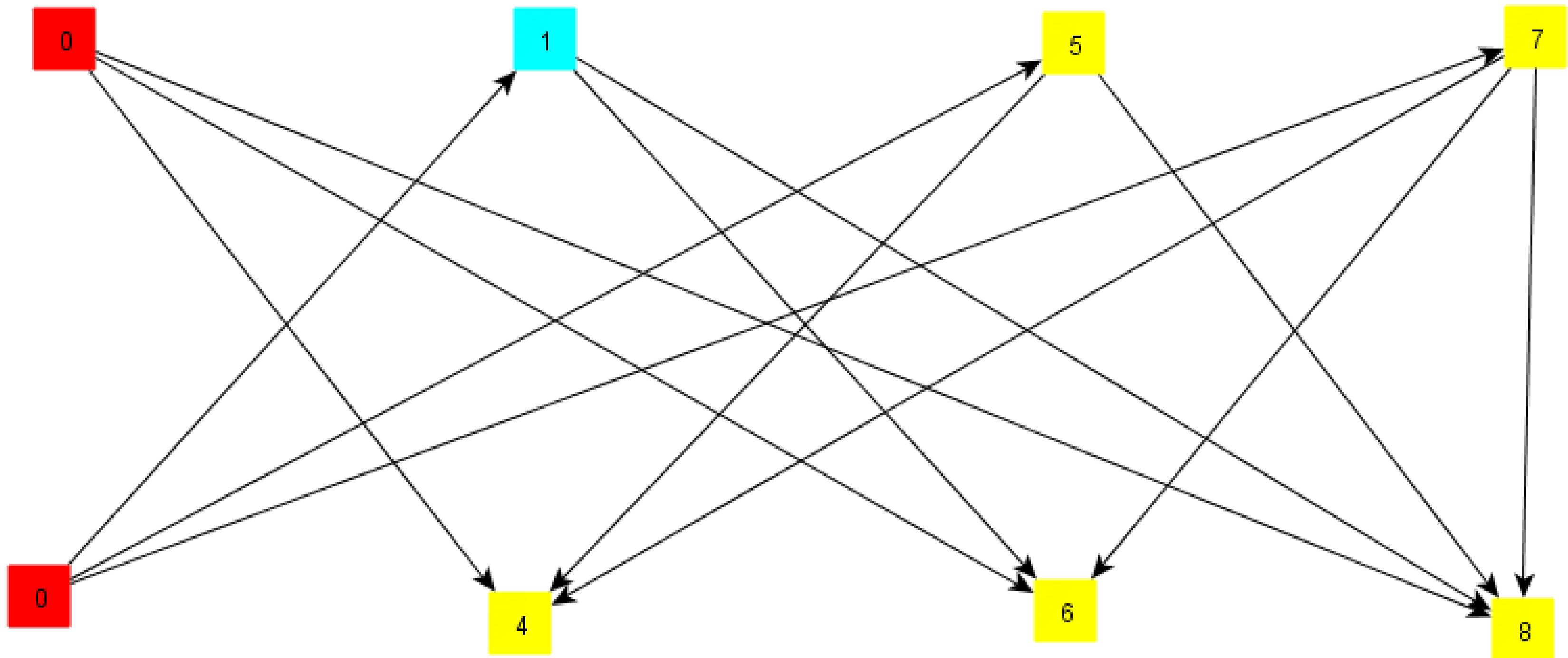
Sequential FF E.g. Step 2

$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}$ $L_5 = \{f, t, t, t\}$ $L_7 = \{f, t, t, t\}$

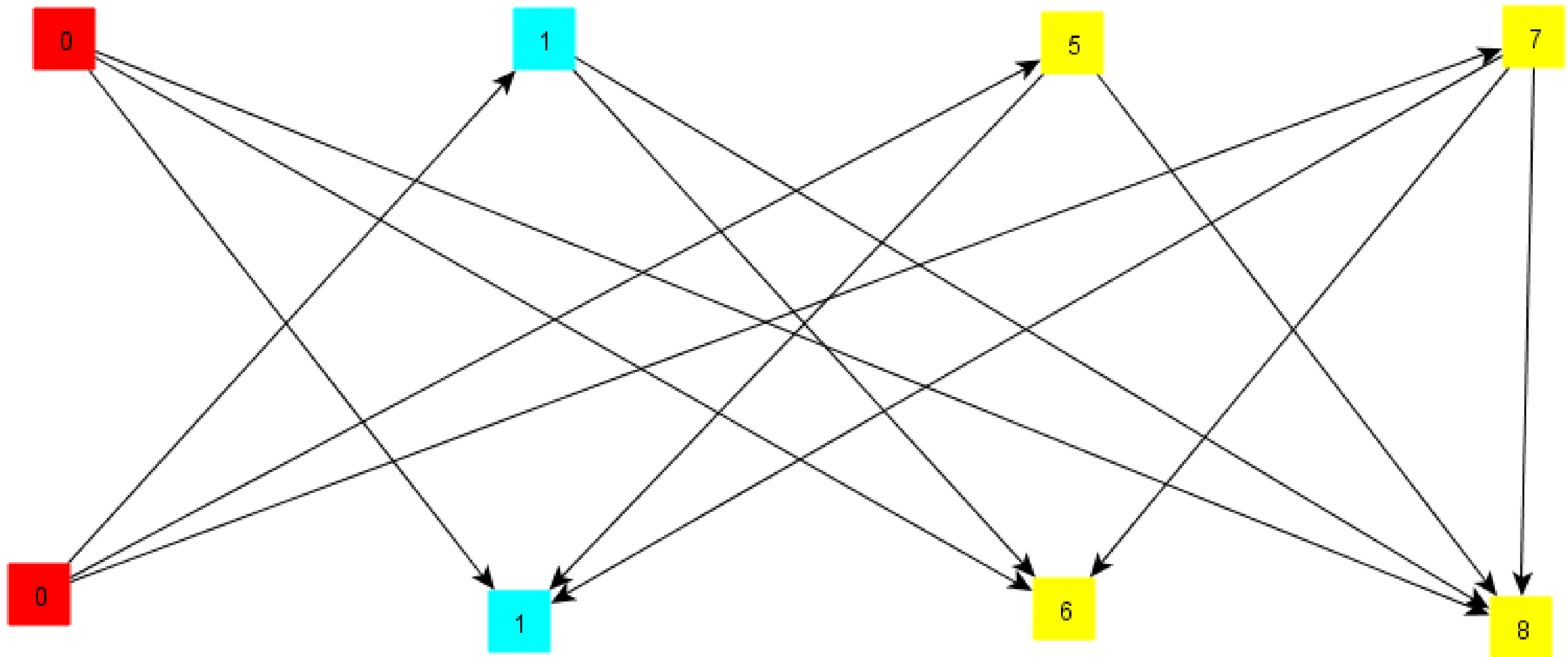


$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$ $L_6 = \{f, t, t, t\}$

Sequential FF E.g. Step 3

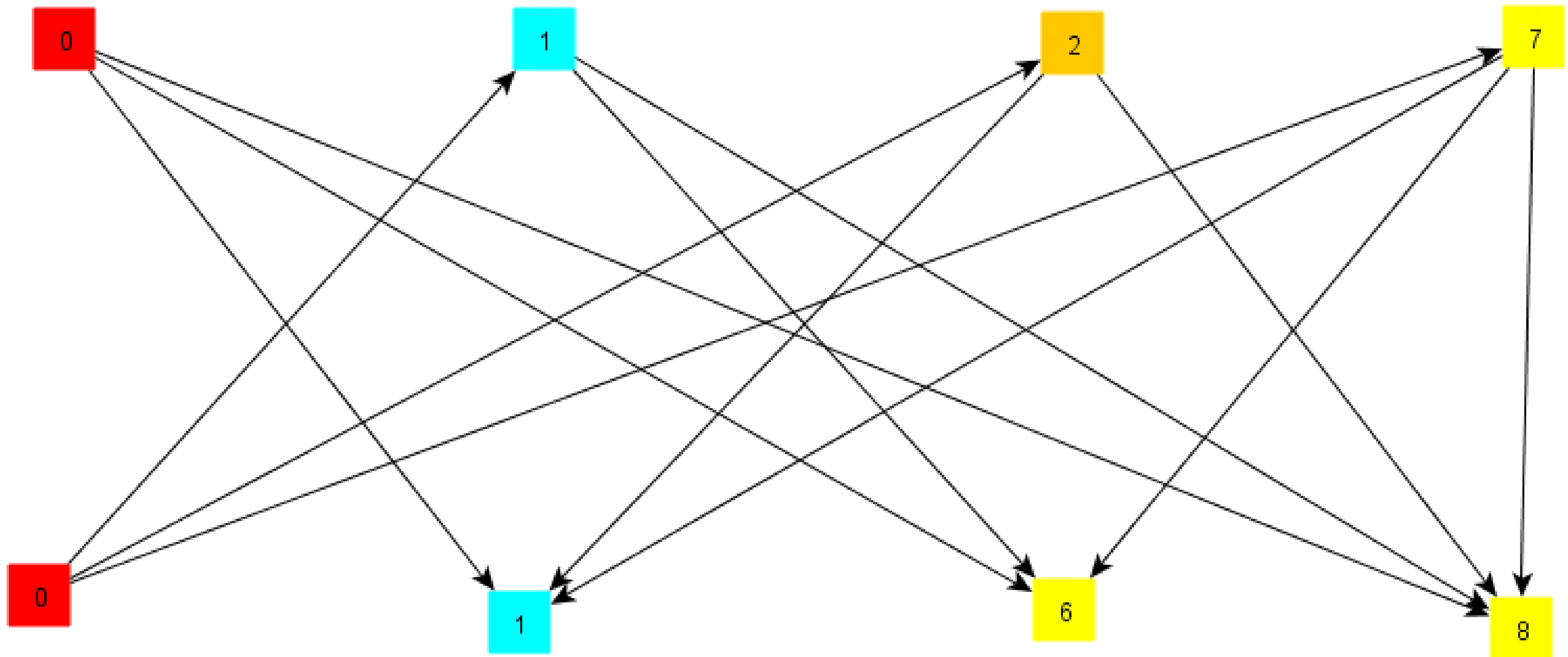


Sequential FF E.g. Step 4



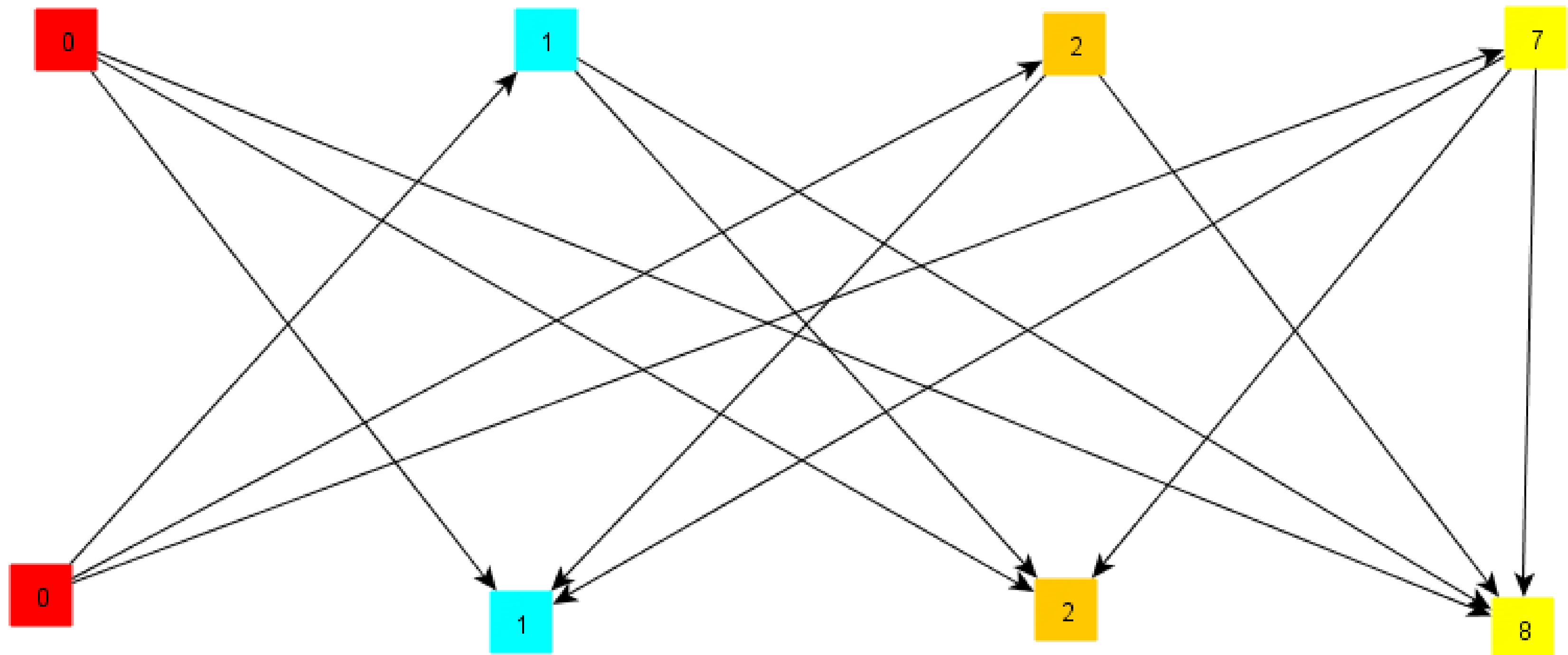
Sequential FF E.g. Step 5

$L_1 = \{t, t, t, t\}$,k=0 $L_3 = \{f, t, t, t\}, k$ =1 $L_5 = \{f, f, t, t\}$,k=2 $L_7 = \{f, f, t, t\}$

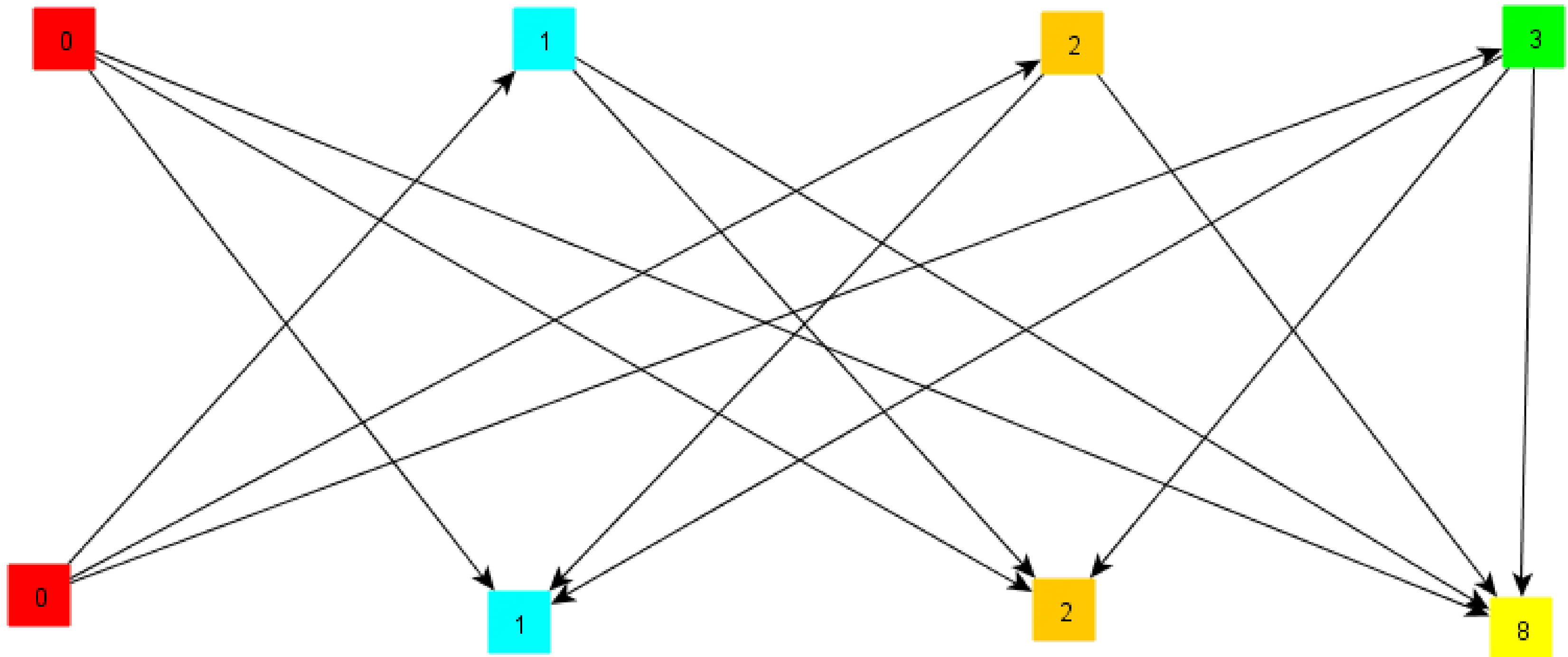


$L_2 = \{t, t, t, t\}$,k=0 $L_4 = \{f, t, t, t\}, \mathrm{k} = 1$ $L_6 = \{f, f, t, t\}$ $L_6 = \{f, f, f, t\}$

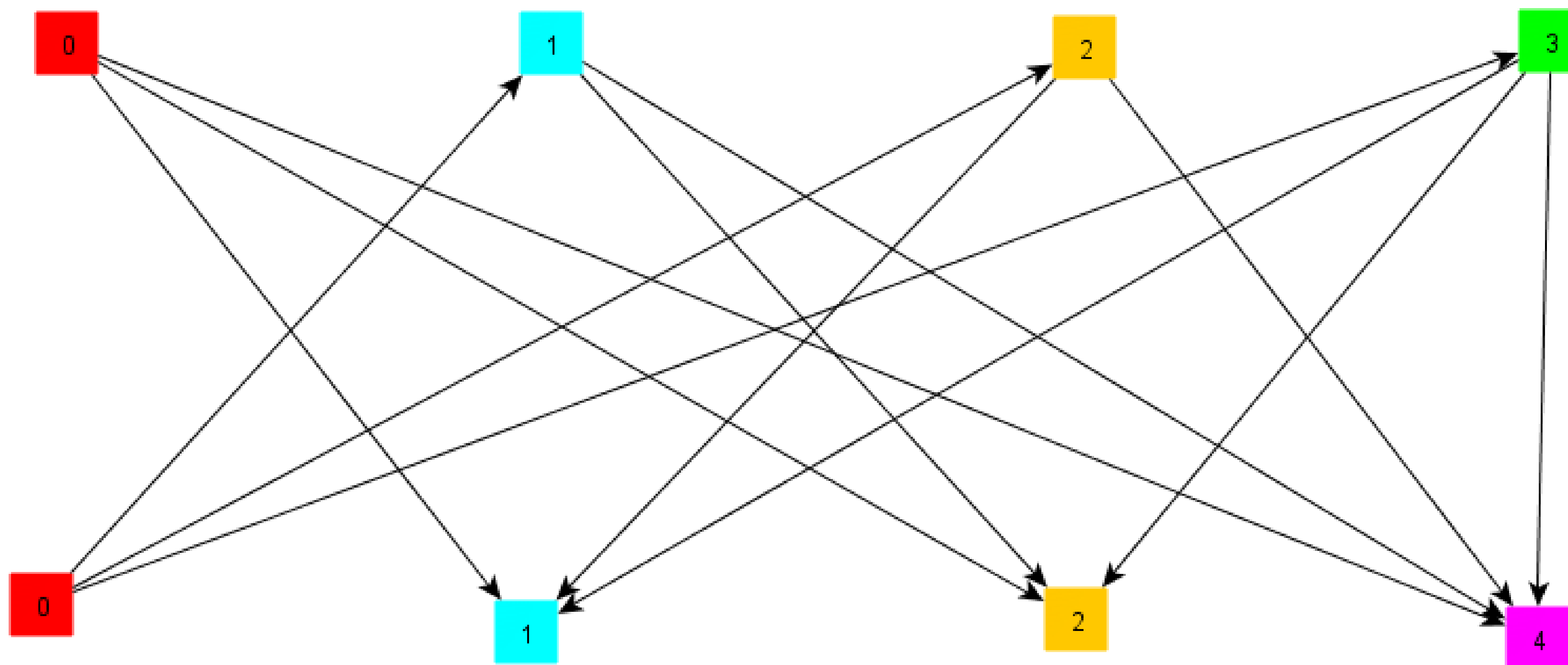
Sequential FF E.g. Step 6



Sequential FF E.g. Step 7



Sequential FF E.g. Step 8



Parallel FF (Subgraph Based)

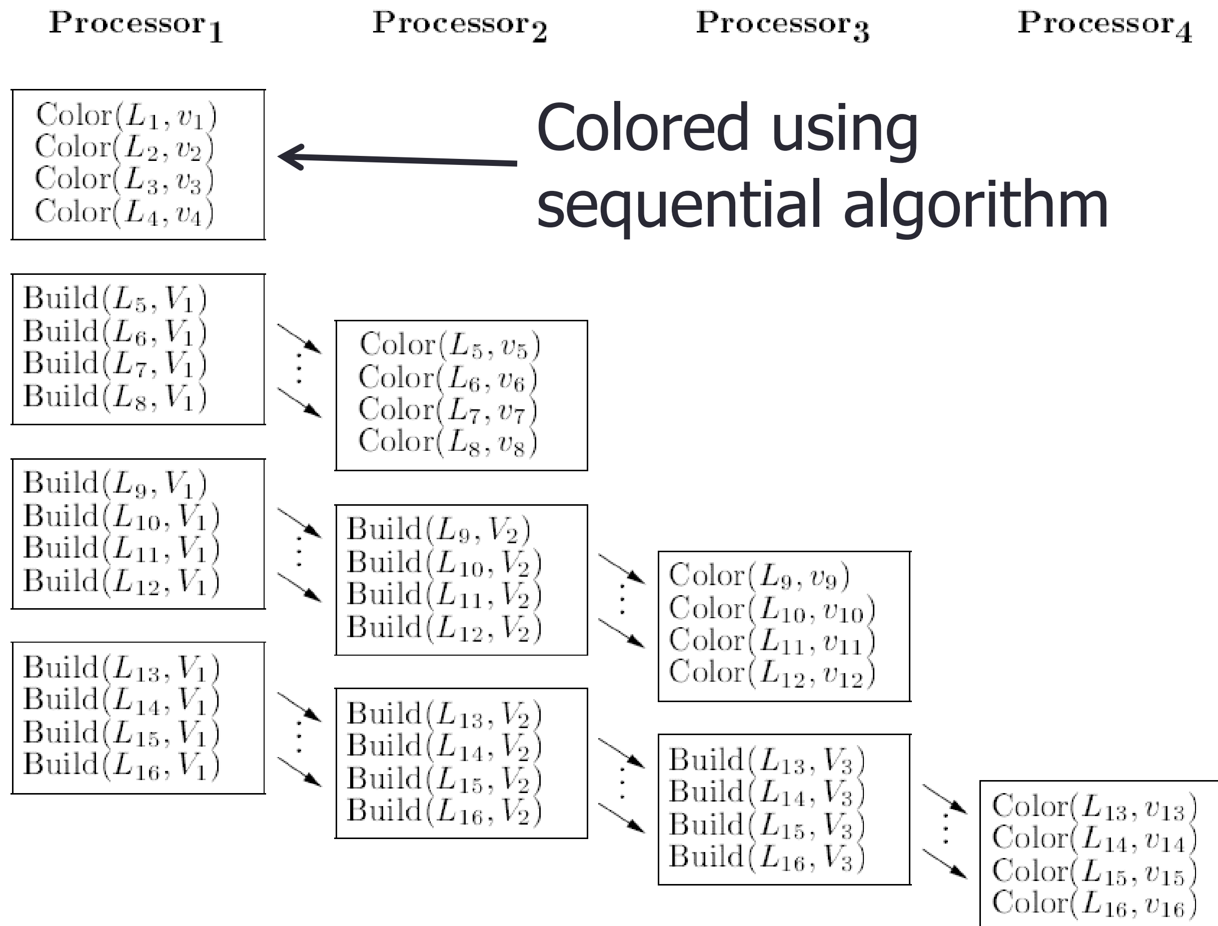
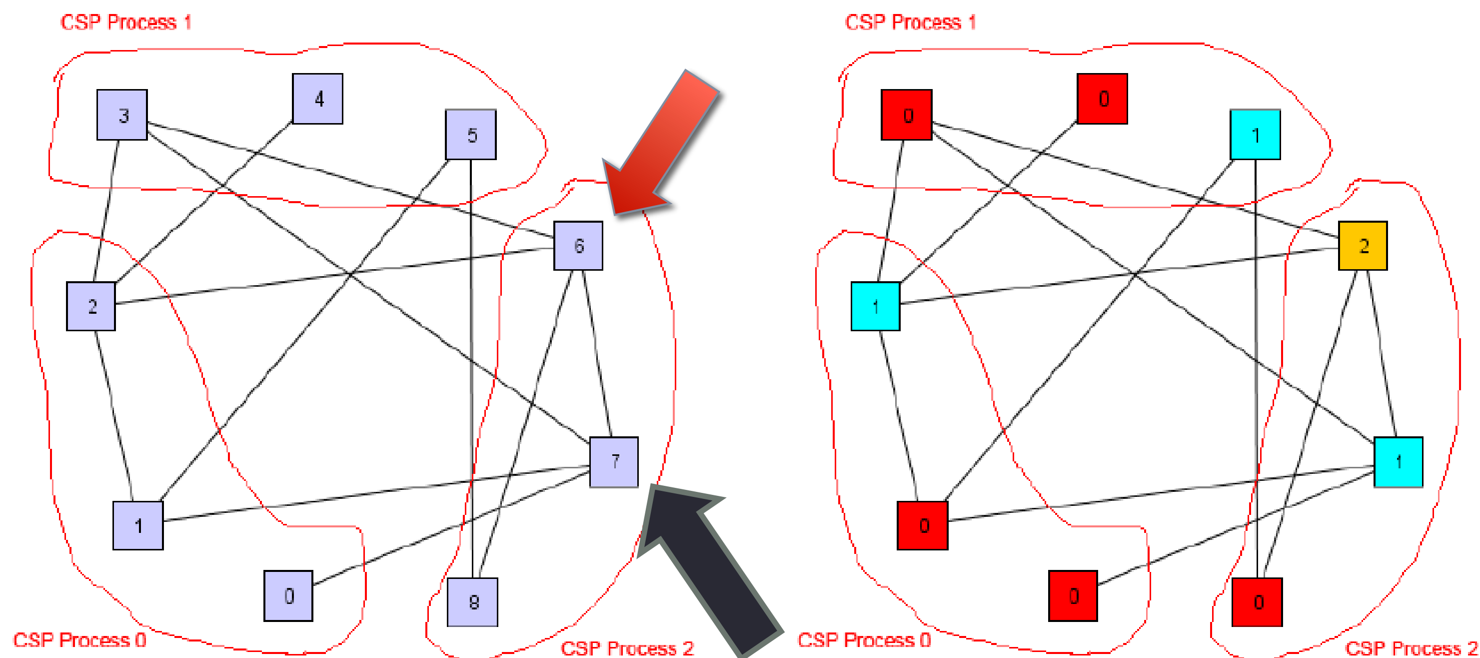


Figure 3: Generalized parallel first fit (16 vertices, 4 processors).

Bug in the Implementation

- JPF found **java.lang.NullPointerException**
- Previously believed to be fixed



(a) Minimal graph producing the bug .

(b) The produced proper coloring.

- Before:
  - Color = Get color from a hashtable
  - Build color for v_i
- After fixing:
  - Color = -1
  - While (Color == -1)
    - Color = Get color from a hashtable
    - Build color for v_i

Other Data Races

- Using **gov.nasa.jpf.listener.PreciseRaceDetector**
- Were not detected
  - Testing with JPF allowed removal of synchronized blocks which were introduced to address the bug mentioned earlier
  - Still not sure if the removal is safe, however, JPF did not detect any issues

Correctness Test Re-visited

- Using JPF to verify proper coloring

```
for each node n in G
    for each neighbour of n
        assert color(n) != color (neighbour)
```

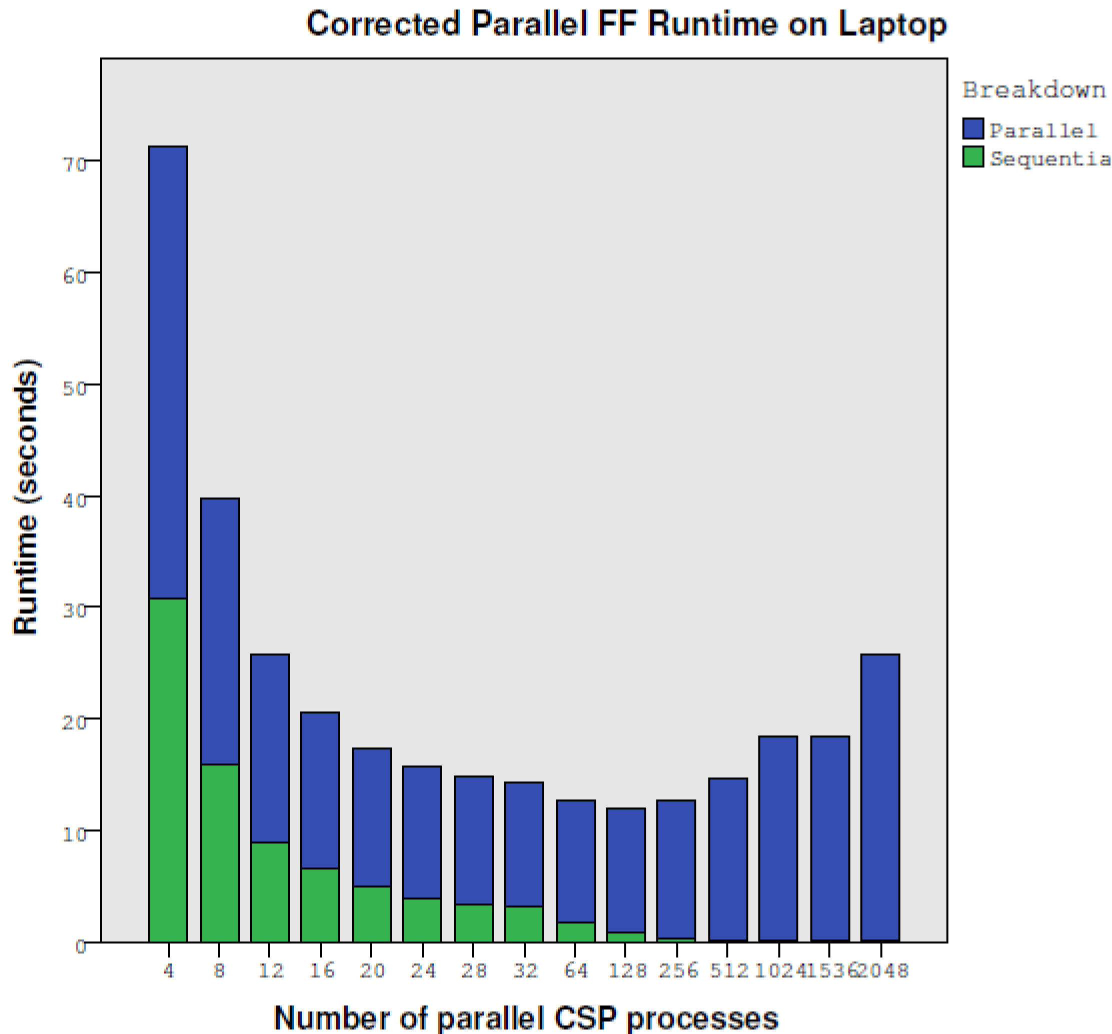
- No assertion violations were detected
- Can’t use too large graphs
  - 2000 nodes, 999001 edges => **java.lang.StackOverflowError**

- **9 nodes 12 edges, 4 proccesses**
- == statistics
- elapsed time: 0:00:03
- states: new=1442, visited=3, backtracked=10, end=8
- search: maxDepth=1434, constraints=0
- choice generators: thread=1435, data=0
- heap: gc=1334, new=1221, free=349
- instructions: 38592
- max memory: 91MB
- loaded code: classes=161, methods=2384

- ==
- **20 nodes 190 edges, 20 processes**
- == statistics
- elapsed time: 0:00:47
- states: new=24802, visited=0, backtracked=7, end=8
- search: maxDepth=24794, constraints=0
- choice generators: thread=24795, data=0
- heap: gc=24247, new=2623, free=1191
- instructions: 327855
- max memory: 453MB
- loaded code: classes=161, methods=2406
- ==

- **40 nodes 400 edges, 20 processes**
- == statistics
- elapsed time: 0:04:23
- states: new=98266, visited=30, backtracked=37, end=8
- search: maxDepth=98258, constraints=0
- choice generators: thread=98259, data=0
- heap: gc=95888, new=4313, free=2347
- instructions: 960172
- max memory: 1361MB
- loaded code: classes=162, methods=2408
- ==

Performance Test Re-visited



- 4 trials
- Same input as before
- New overhead of **while** loop
- Removed overhead of synched blocks
- Output slightly improved overall, but it could be due to chance as well

Bug in JCSP

- Found **gov.nasa.jpf.jvm.NotDeadlockedProperty**
- === snapshot #1
- thread index=1,name=ParallelFirstFit$FFGeneralProcess@a998,status=WAITING,this=org.jcsp.lang.ParThread@667,waiting on: java.lang.Object@679
- call stack:
- at org.jcsp.lang.Barrier.sync(Barrier.java:33)
- at org.jcsp.lang.ParThread.run(ParThread.java:41)
- …
- The same bug was observed for a simple single One2One Channel, 2-thread reader and writer JCSP application

Conclusion & Open Questions

- Conclusion
  - JPF can be ful for finding issues in CSP
    - But misleading when detecting deadlocks due to a bug
- Is it reasonable to keep increasing the size of input and number of processes for this kind of algorithm?
  - The bug was caught using only a 9 node and 12 edges graph among 4 processes
  - Limits on how large the input can get, particular the edges
- Will different search strategies help?
  - Currently only the default strategy was tried