Why generate problems?
- Finding an optimal solution to the MSP is NP-hard
- But, we were stubborn…
  - observed most MSPs in a high-level synthesis context can be solved with an exact, ILP-based scheduler
  - only a handful instances are slow or intractable, too few to reason about
- Generated problems “fill the gaps” between the benchmark instances
  - small/large, sparse/dense, few/many limited operations, …
  - investigate what’s “hard” for a particular scheduler
  - long-term goal: build an oracle that picks the “right” scheduler for a given instance

Formal definition
An instance of the modulo scheduling problem (MSP) is defined by:
- Resources types \( r \)
  - latency
  - # available units (or \( \infty \))
- Operations \( i \)
  - mapping to resource type
- Edges \( i \to j \)
  - delay (e.g. to control operator chaining)
  - distance (\( \geq 1 \) for inter-iteration dependences)
- Solution: initiation interval (II), start times for operations

Generating graphs with known MinII
- MinII = lower bound for optimal II, induced by cycles and resource constraints
  - schedulers usually try several candidate IIs until a feasible solution is found
  - important to keep number of tried candidate IIs the same when comparing scheduler runtimes
- GeMS allows a desired MinII, and whether the MSP shall be feasible or infeasible at that MinII, to be specified
  - if needed, picks operations to construct a cycle (step 3) to raise the graph’s MinII
  - checks prevent that edges (generated in step 4) change the desired MinII or its feasibility
  - the rest of the MSP is still randomly generated

Code example
Resource resA = new Resource("A", 2, 2); Resource resB = new Resource("B", 1, 4); Resource resC = new Resource("C", 8);
GraphGenerator gen = new GraphGenerator(
  new FixedShapeLayerCreator(# nodes in layer \( \leq 1, 2, 4, 1 \)),
  new DistributionNodeCreator(new ProbabilityDistribution<>(resA, resB, resC)),
  new EdgeCreator(
    // new ConstantValueComputer(0),
    // new ConstantValueComputer(8),
    // new ConstantValueComputer(1),
    // new ConstantValueComputer(11),
    new ProbabilityEdgeIncluder(0.8038),
    new ProbabilityEdgeIncluder(0.8075),
    new ProbabilityEdgeIncluder(0.8075),
    new ProbabilityEdgeIncluder(0.8875),
    new ProbabilityEdgeIncluder(0.8875)
);,
GraphFileUtils.graphToHatScheTFiles(gen.createGraph(# seed \( \leq 42 \)), "graph");
- GeMS is a toolkit written in Java, offers no CLI
- Graph representation is simple (~nodes+edges)
  - supplied export facilities: DOT, and format used by HatScheT scheduler library

Case study
- Question: How does the Moovac formulation [CASES’16] cope with symmetry?
- Experiment
  - 1 resource type with 2 instances
  - 48 operations in different layer structures compete for this resource type
- Result/insight
  - the more operations in parallel, the harder for Moovac to find/prove an optimal solution

<table>
<thead>
<tr>
<th>Layers x Ops</th>
<th>48x1</th>
<th>24x2</th>
<th>16x3</th>
<th>12x4</th>
<th>8x6</th>
<th>6x8</th>
<th>4x12</th>
<th>2x24</th>
<th>1x48</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg time [s]</td>
<td>3.0</td>
<td>116.5</td>
<td>3600</td>
<td>3600</td>
<td>3600</td>
<td>3600</td>
<td>3600</td>
<td>3600</td>
<td>3600</td>
</tr>
<tr>
<td>avg gap [%]</td>
<td>opt.</td>
<td>opt.</td>
<td>29</td>
<td>45</td>
<td>61</td>
<td>69</td>
<td>77</td>
<td>88</td>
<td>88</td>
</tr>
</tbody>
</table>

Average over 10 random instances ✤ No solution found for 2 instances

Outlook
- Add support for specifying the number of incoming edges (e.g. #operands)
- Finer control over the MSP’s II (e.g. “be feasible at MinII+3”)