Constraint Streams 1-01

The future of score constraints in OptaPlanner

Lukáš Petrovický
Principal Software Engineer

September 2nd, 2020

Red Hat

Agenda

- Motivation
- Basic concepts
  - Filtering,
  - Expansion,
  - Transformation.
- Testing

Note: Some prior knowledge of *OptaPlanner* required.

Why *Constraint Streams*?

- Java developers don't want to learn DRL.
- "Easy" Java score calculator too slow.
  - No support for score explanation.
- "Incremental" Java score calculator too hard to write.
- **The ideal middle ground**:
  - Written in plain Java,
  - performance on par with DRL,
  - score explanation capable,
  - with unit testing tools available.

CS Anatomy: Constraint Provider

```
public class CloudBalancingConstraintProvider implements ConstraintProvider {

    @Override
    public Constraint[] defineConstraints(ConstraintFactory constraintFactory) {
        return new Constraint[] {
                requiredCpuPowerTotal(constraintFactory),
                requiredMemoryTotal(constraintFactory),
                requiredNetworkBandwidthTotal(constraintFactory),
                computerCost(constraintFactory)
        };
    }
```

CS Anatomy: Constraint Provider

```
public class CloudBalancingConstraintProvider implements ConstraintProvider {

    @Override
    public Constraint[] defineConstraints(ConstraintFactory constraintFactory) {
        return new Constraint[] {
                requiredCpuPowerTotal(constraintFactory),
                requiredMemoryTotal(constraintFactory),
                requiredNetworkBandwidthTotal(constraintFactory),
                computerCost(constraintFactory)
        };
    }
```

CS Anatomy: Simplest possible Constraint

```
Constraint computerCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class)
            .penalize( constraintName: "computerCost",
                    HardSoftScore.ONE_SOFT,
                    CloudComputer::getCost);
}
```

CS Anatomy: Simplest possible Constraint

```
Constraint computerCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class)
            .penalize( constraintName: "computerCost",
                    HardSoftScore.ONE_SOFT,
                    CloudComputer::getCost);
}
```

My first CS: `from(Something.class)`

- Operates on **planning entities** and **problem facts** from planning solution. (“Facts.”)
- Returns a *Constraint Stream* of facts whose type matches the one requested.
- If the facts are planning entities, it will **only return the ones already initialized** by a construction heuristic.

My first CS: Applying penalties

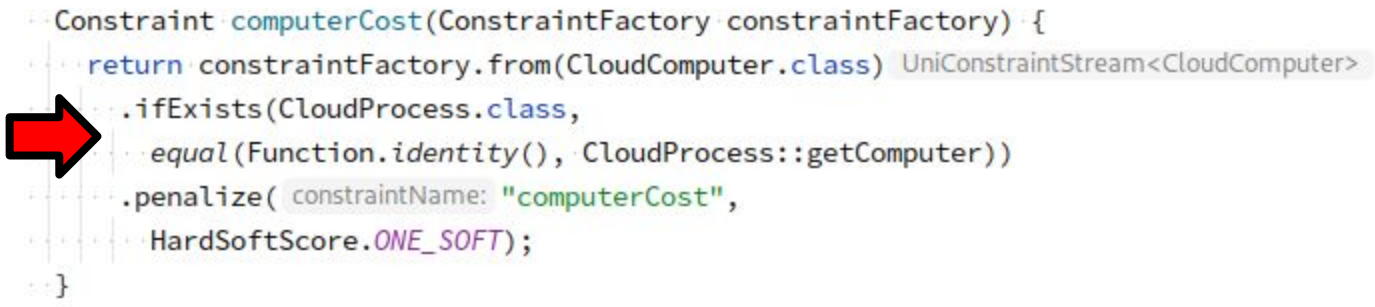
- Three parts to a `penalize(...)` call:
  - Unique constraint name. (“computerCost”)
  - **Constraint weight**. (`HardSoftScore.ONE_SOFT`)
    - A constant describing how heavy the penalty should be.
  - **Match weight**. (`CloudComputer::getCost`)
    - Multiplier derived from the matching fact.
- **Total penalty: Constraint Weight × Match Weight.**
  - Computer with cost 2 impacts score by -2soft.
- Use `reward(...)` for positive score impact instead.

Filtering stream contents

```
Constraint computerCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class) UniConstraintStream<CloudComputer>
        .filter(computer -> computer.getCost() > 5)
        .penalize( constraintName: "computerCost",
            HardSoftScore.ONE_SOFT);
}
```

- Only `CloudComputers` with `cost > 5` will be penalized.
- The score impact will be -1soft for every such `CloudComputer`.

Filtering stream contents using context



```
Constraint computerCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class) UniConstraintStream<CloudComputer>
            .ifExists(CloudProcess.class,
                    equal(Function.identity(), CloudProcess::getComputer))
            .penalize( constraintName: "computerCost",
                    HardSoftScore.ONE_SOFT);
}
```

- Only penalize `CloudComputer` if it has a `CloudProcess` running on it.

`ifExists(SomethingElse.class)`

- Only propagates a fact if some other fact is available.
- The condition is met when **even just a single fact** of type `SomethingElse` is found.
  - The fact itself **can not be accessed further downstream**.
- Use `ifNotExists(...)` for the opposite effect.

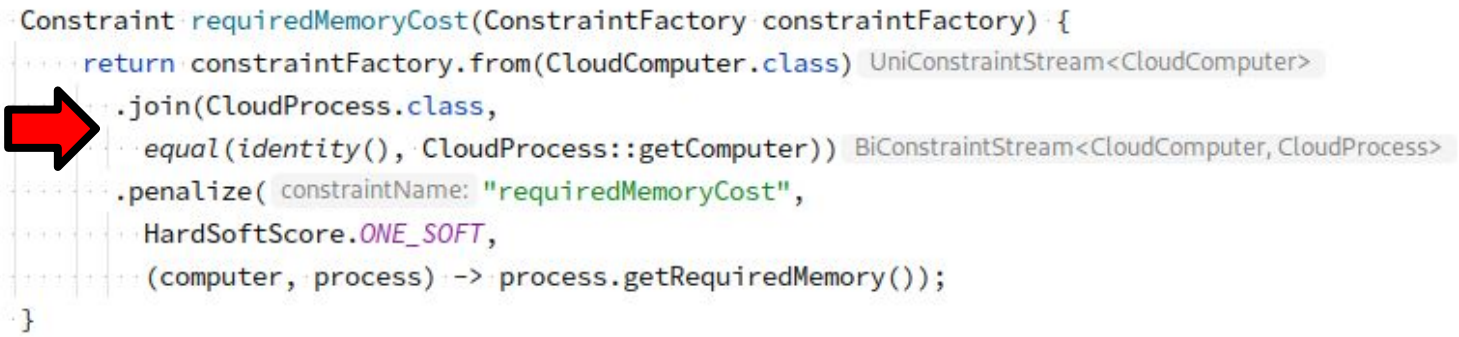
Joiners

- Specify a fact's intended relation to another fact.
  - `Joiners.equal(A⇒X1, B⇒X2)` will match when `X1.equals(X2)`.
- In our example:
  - `computer -> computer`
    - `CloudComputer ⇒ CloudComputer`,
  - `CloudProcess::getComputer`
    - `CloudProcess ⇒ CloudComputer`
  - Joiner will match when `process.getComputer()` points to the same `CloudComputer` coming from upstream.

Out-of-the-box joiners

- `Joiners.equal()`,
- `Joiners.greaterThan()`, `lessThan()`, ...
  - For objects implementing `Comparable`.
- `Joiners.filtering(...)` for when other joiners are insufficient
  - Avoid if you can. Other **joiners allow for better performance** through indexing.
  - Other joiners are also preferable to standalone `filter(...)` calls.
- See `org.optaplanner.core.api.score.stream.Joiners`.

Expanding the stream



```
Constraint requiredMemoryCost(ConstraintFactory constraintFactory) {
        return constraintFactory.from(CloudComputer.class) UniConstraintStream<CloudComputer>
                .join(CloudProcess.class,
                        equal(identity(), CloudProcess::getComputer)) BiConstraintStream<CloudComputer, CloudProcess>
                .penalize( constraintName: "requiredMemoryCost",
                        HardSoftScore.ONE_SOFT,
                        (computer, process) -> process.getRequiredMemory());
}
```

- Penalize every `CloudProcess`.
- Score impact will be proportional to how much memory the process requires.

Expanding the stream

```
Constraint requiredMemoryCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class) UniConstraintStream<CloudComputer>
        .join(CloudProcess.class,
            equal(identity(), CloudProcess::getComputer)) BiConstraintStream<CloudComputer, CloudProcess>
        .penalize( constraintName: "requiredMemoryCost",
            HardSoftScore.ONE_SOFT,
            (computer, process) -> process.getRequiredMemory());
}
```

- Penalize every `CloudProcess`.
- Score impact will be proportional to how much memory the process requires.

Expanding the stream

```
Constraint requiredMemoryCost(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudComputer.class) UniConstraintStream<CloudComputer>
        .join(CloudProcess.class,
            equal(identity(), CloudProcess::getComputer)) BiConstraintStream<CloudComputer, CloudProcess>
        .penalize( constraintName: "requiredMemoryCost",
            HardSoftScore.ONE_SOFT,
            (computer, process) -> process.getRequiredMemory());
}
```

- Penalize every `CloudProcess`.
- Score impact will be proportional to how much memory the process requires.

Increasing stream cardinality with `join(....)`

- **`join(...)` makes the matched facts available downstream,** increasing stream cardinality.
- `from(A).join(B, ...)` **creates a cartesian product** of A and B
  - With 2 A-facts and 3 matching B-facts, **6 unique pairs of `(A,B)`** are sent downstream.
  - If an A-fact has no matching B-facts, it is not sent downstream.
  - `Joiners` apply just as they do with `ifExists(...)` to reduce the number of matching facts.

Stream cardinality

- `from(A)`
  - `UniConstraintStream<A>`.
  - 1-element tuples.
  - **`Function<A,...>`; `Predicate<A>`**
- `from(A).join(B)`
  - `BiConstraintStream<A,B>`.
  - 2-element tuples.
  - **`BiFunction<A,B,...>`; `BiPredicate<A, B>`**
- Can join all the way up to `QuadConstraintStream<A,B,C,D>`.

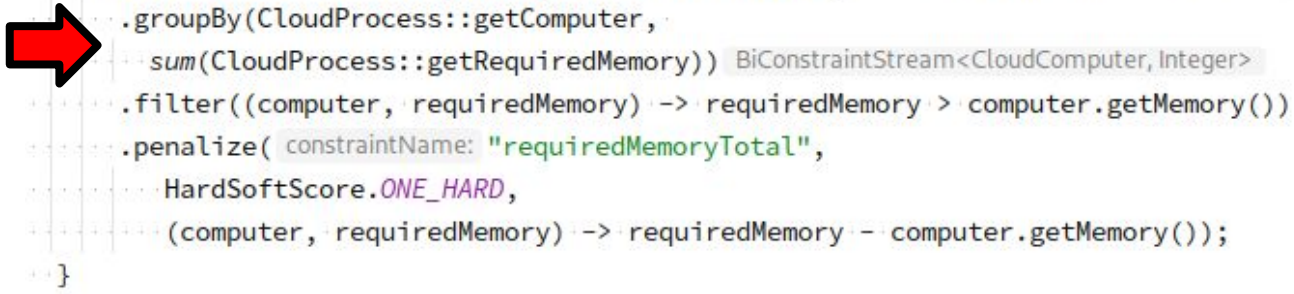
Stream cardinality ctd.

- **Prefer `ifExists(...)` to `join(...)`** unless you need downstream access to the matched facts
  - `ifExists(...)` does not create cartesian products, and therefore performs better.
- If you must use `join(...)`, **be as specific with your joiners as you can** to limit the cartesian product.

Recap so far

- Simple penalties and rewards
  - `from(...)`,
  - `penalize(...)`, `reward(...)`.
- Filtering the stream
  - `filter(...)`,
  - `ifExists(...)`, `ifNotExists(...)`.
- Stream cardinality
  - `join(...)`.

Transforming the stream



```
Constraint requiredMemoryTotal(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudProcess.class) UniConstraintStream<CloudProcess>
            .groupBy(CloudProcess::getComputer,
                    sum(CloudProcess::getRequiredMemory)) BiConstraintStream<CloudComputer, Integer>
            .filter((computer, requiredMemory) -> requiredMemory > computer.getMemory())
            .penalize( constraintName: "requiredMemoryTotal",
                    HardSoftScore.ONE_HARD,
                    (computer, requiredMemory) -> requiredMemory - computer.getMemory());
}
```

- Add a hard penalty for every computer where processes take up more memory than is available.

Transforming the stream

```
Constraint requiredMemoryTotal(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudProcess.class) UniConstraintStream<CloudProcess>
            .groupBy(CloudProcess::getComputer,
                    sum(CloudProcess::getRequiredMemory)) BiConstraintStream<CloudComputer, Integer>
            .filter((computer, requiredMemory) -> requiredMemory > computer.getMemory())
            .penalize( constraintName: "requiredMemoryTotal",
                    HardSoftScore.ONE_HARD,
                    (computer, requiredMemory) -> requiredMemory - computer.getMemory());
}
```

- Add a hard penalty for every computer where processes take up more memory than is available.

Transforming the stream

```
Constraint requiredMemoryTotal(ConstraintFactory constraintFactory) {
    return constraintFactory.from(CloudProcess.class) UniConstraintStream<CloudProcess>
            .groupBy(CloudProcess::getComputer,
                    sum(CloudProcess::getRequiredMemory)) BiConstraintStream<CloudComputer, Integer>
            .filter((computer, requiredMemory) -> requiredMemory > computer.getMemory())
            .penalize( constraintName: "requiredMemoryTotal",
                    HardSoftScore.ONE_HARD,
                    (computer, requiredMemory) -> requiredMemory - computer.getMemory());
}
```

- Add a hard penalty for every computer where processes take up more memory than is available.

The power of `groupBy(...)`

- The only construct in *Constraint Streams* that **allows you to create information not already available** in the planning solution.
- Takes all upstream tuples and creates a new transformed stream, likely with a different number of tuples and different cardinality.

Constraint collectors

- Reduce a **group of tuples into a single result.**
- Assume stream of `CloudProcess`
  - `count(CloudProcess::getComputer)` returns a total number of running computers.
  - `sum(CloudProcess::getRequiredMemory)` returns the total memory required by running processes.
- See `org.optaplanner.core.api.score.stream.ConstraintCollectors` .

Group key mapping

- Turns a set of facts into a new set of unique facts.
- Example:
  - Assume stream of `CloudProcess`
  - Mapping: `CloudProcess::getComputer`
  - Input:
    - Process1 @ Computer1
    - Process2 @ Computer2
    - Process3 @ Computer1
  - Output:
    - `Computer1`; `Computer2`

Group key mapping ctd.

Another example:

- Mapping:
  - `CloudProcess::getComputer`
  - `CloudProcess::getRequiredMemory`
- Input:
  - Process1 @ Computer1, Required Memory 1
  - Process2 @ Computer2, Required Memory 4
  - Process3 @ Computer1, Required Memory 3
  - Process4 @ Computer1, Required Memory 3
- Output:
  - `(Computer1, 1); (Computer2, 4); (Computer1, 3)`

Putting it all together

```
groupBy(CloudProcess::getComputer,
    sum(CloudProcess::getRequiredMemory))
```

1. Takes **all** `CloudProcess` **instances**,
2. separates them into **groups** where they **share** the **same** `CloudComputer`,
3. calculates a **sum of** memory required by **this group** of processes,
4. returns `Bi...Stream<CloudComputer, Integer>`.

Testing the constraint

```
private final ConstraintVerifier<CloudBalancingConstraintProvider, CloudBalance> constraintVerifier =
    ConstraintVerifier.build(new CloudBalancingConstraintProvider(), CloudBalance.class, CloudProcess.class);

@Test
public void computerCost() {
    CloudComputer computer1 =
    CloudComputer computer2 =
    CloudProcess unassignedProcess =
    CloudProcess process =
        process.setComputer(computer1);

    constraintVerifier.verifyThat(CloudBalancingConstraintProvider::computerCost) SingleConstraintVerification<CloudBalance>
        .given(computer1, computer2, unassignedProcess, process) SingleConstraintAssertion
        .penalizesBy( matchWeightTotal: 2);
}
```

- Constraint weights are not tested, as they are externally configurable.

Testing the constraint

```
private final ConstraintVerifier<CloudBalancingConstraintProvider, CloudBalance> constraintVerifier =
    ConstraintVerifier.build(new CloudBalancingConstraintProvider(), CloudBalance.class, CloudProcess.class);

@Test
public void computerCost() {
    CloudComputer computer1 =
    CloudComputer computer2 =
    CloudProcess unassignedProcess =
    CloudProcess process =
        process.setComputer(computer1);

    constraintVerifier.verifyThat(CloudBalancingConstraintProvider::computerCost) SingleConstraintVerification<CloudBalance>
        .given(computer1, computer2, unassignedProcess, process) SingleConstraintAssertion
        .penalizesBy( matchWeightTotal: 2);
}
```

- Constraint weights are not tested, as they are externally configurable.

Constraint Streams: One API to rule them all?

	DRL	*Plain Java*		
	Drools	**Easy**	**Incremental**	***Constraint Streams***
Learning Curve	Steep	None	OK	OK
Ease of use	OK	Good	Poor	Good
Ease of testing	OK	OK	Poor	Good
Performance	OK	Poor	Excellent	OK

- No plans for deprecating any of these,
- but *Constraint Streams* is **quickly becoming the default**.

Assumes a typical Java developer without prior knowledge of either Drools or *Constraint Streams*.
Performance is considered in relation to the other methods.

Find out more

- See documentation:
  - optaplanner.org/learn/documentation.html
- See `ConstraintProvider` implementations (and tests) on Github:
  - github.com/kiegroup/optaplanner
- Ask a question on StackOverflow:
  - stackoverflow.com/questions/tagged/optaplanner
- Join the community on Zulip chat:
  - kie.zulipchat.com

Thank you!

Any questions?


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat