Red Hat JBoss BRMS 6.4

Business Resource Planner ガイド

JBoss 開発者向け

Last Updated: 2020-05-19

Red Hat JBoss BRMS 6.4 Business Resource Planner ガイド

JBoss 開発者向け

Red Hat Customer Content Services
brms-docs@redhat.com

Emily Murphy

Gemma Sheldon

Michele Haglund

Mikhail Ramendik

Stetson Robinson

Vidya Iyengar

法律上の通知

概要

本書では、Business Resource Planner のインストールの手順および使用方法を説明します。

目次

第1章 BUSINESS RESOURCE PLANNER の概要

1.1. BUSINESS RESOURCE PLANNER について

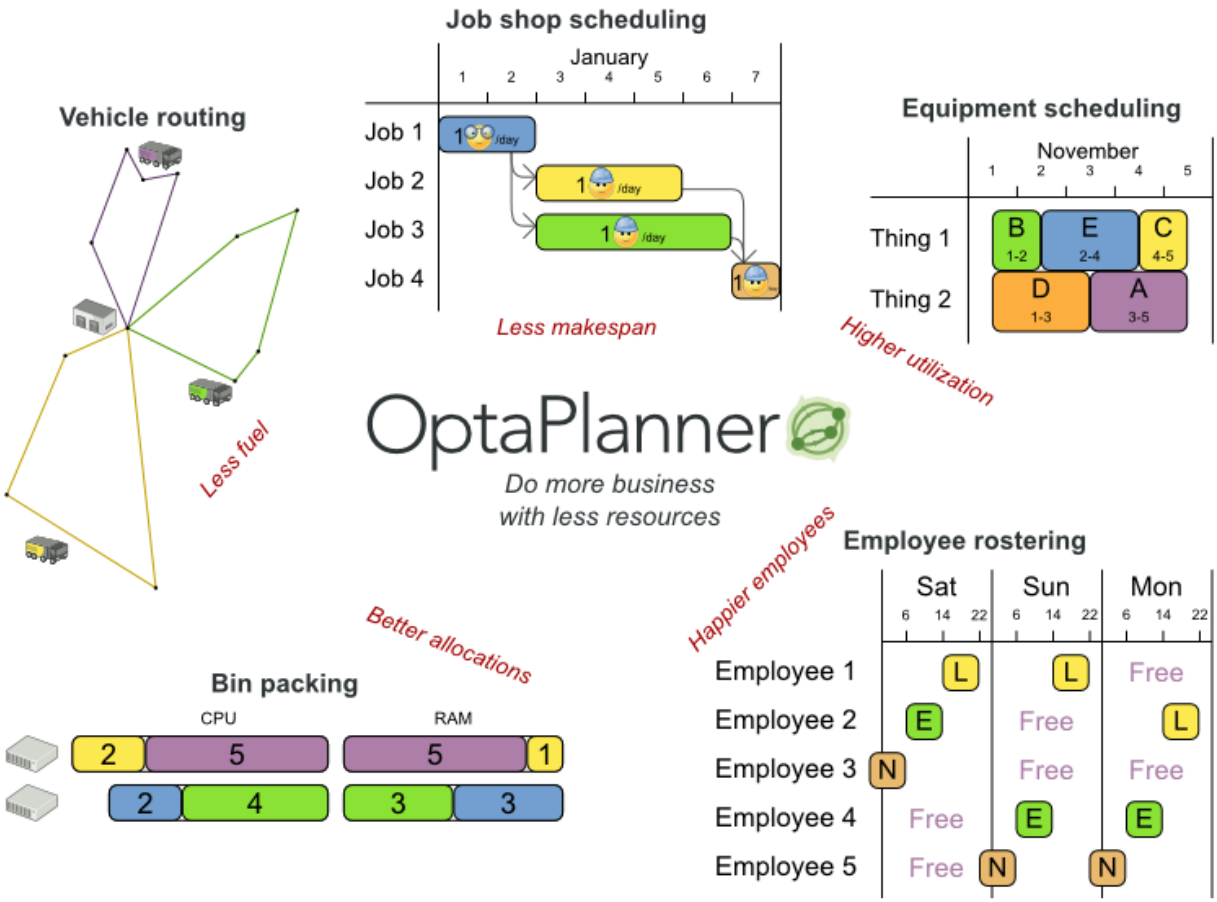
Business Resource Planner は、計画問題を最適化する軽量かつ組み込み可能な計画エンジンです。一般的な Java™ プログラマーが計画問題を効率的に解決できるようにし、最適化ヒューリスティックおよびメタヒューリスティックと、大変効率的なスコア計算を組み合わせます。

Business Resource Planner は、次のような各種ユースケースの解決に役立ちます。

- 従業員勤務表/患者一覧: 看護師の勤務シフトの作成や病床管理の追跡を容易にします。
- 教育機関の時間割: 授業、コース、試験、および会議の計画を容易にします。
- 工場の計画: 自動車の組み立てライン、機械の待機計画、および作業員のタスク計画を追跡します。
- 在庫の削減: 紙や金属などの資源の消費を減らし、無駄を最小限に抑えます。

どの組織も、リソース (従業員、資産、時間、資金) に制約がある中で製品やサービスを提供するといった計画問題に直面しています。

図1.1 ユースケースの概要



Business Resource Planner は、Apache Software License 2.0 を使用するオープンソースソフトウェアです。Planner は、100% pure Java™ に認定されており、どの Java 仮想マシンでも稼働します。

1.2. BUSINESS RESOURCE PLANNER のダウンロード

Business Resource Planner はすぐに実稼働環境で利用できます。API は安定していますが、後方互換性に関する変更が加えられる可能性があります。レシピファイル UpgradeFromPreviousVersionRecipe.txt を使用すれば、新規バージョンに簡単にアップグレードし、後方互換性に関する変更にも素早く対応することができます。このレシピファイルは、全リリースに含まれます。

手順: リリース **ZIP** の取得、およびサンプルの実行

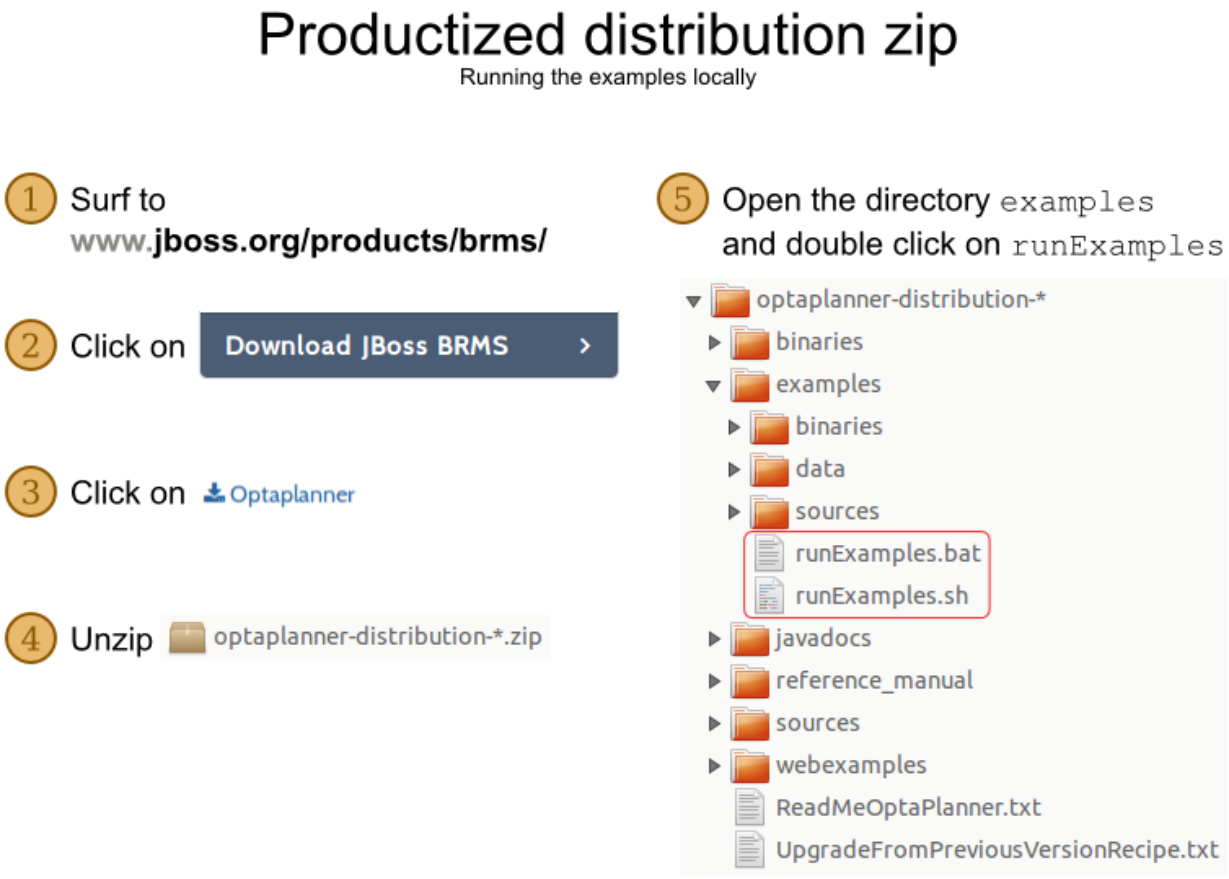
1. Red Hat カスタマーポータル に移動し、ユーザー認証情報でログインします。
2. **Downloads** → **Red Hat JBoss Middleware** → **Download Software** を選択します。
3. **Products** ドロップダウンメニューから **BPM Suite** または **BRMS Platform** を選択します。
4. **Version** のドロップダウンメニューから製品バージョン **6.4** を選択します。
5. **Standalone** または **Deployable** ファイルを選択して、**download** をクリックします。
6. ファイルを展開します。
7. **examples** ディレクトリーを開き、以下のスクリプトを実行します。
   Linux または Mac:

   ```
   $ cd examples
   $ ./runExamples.sh
   ```

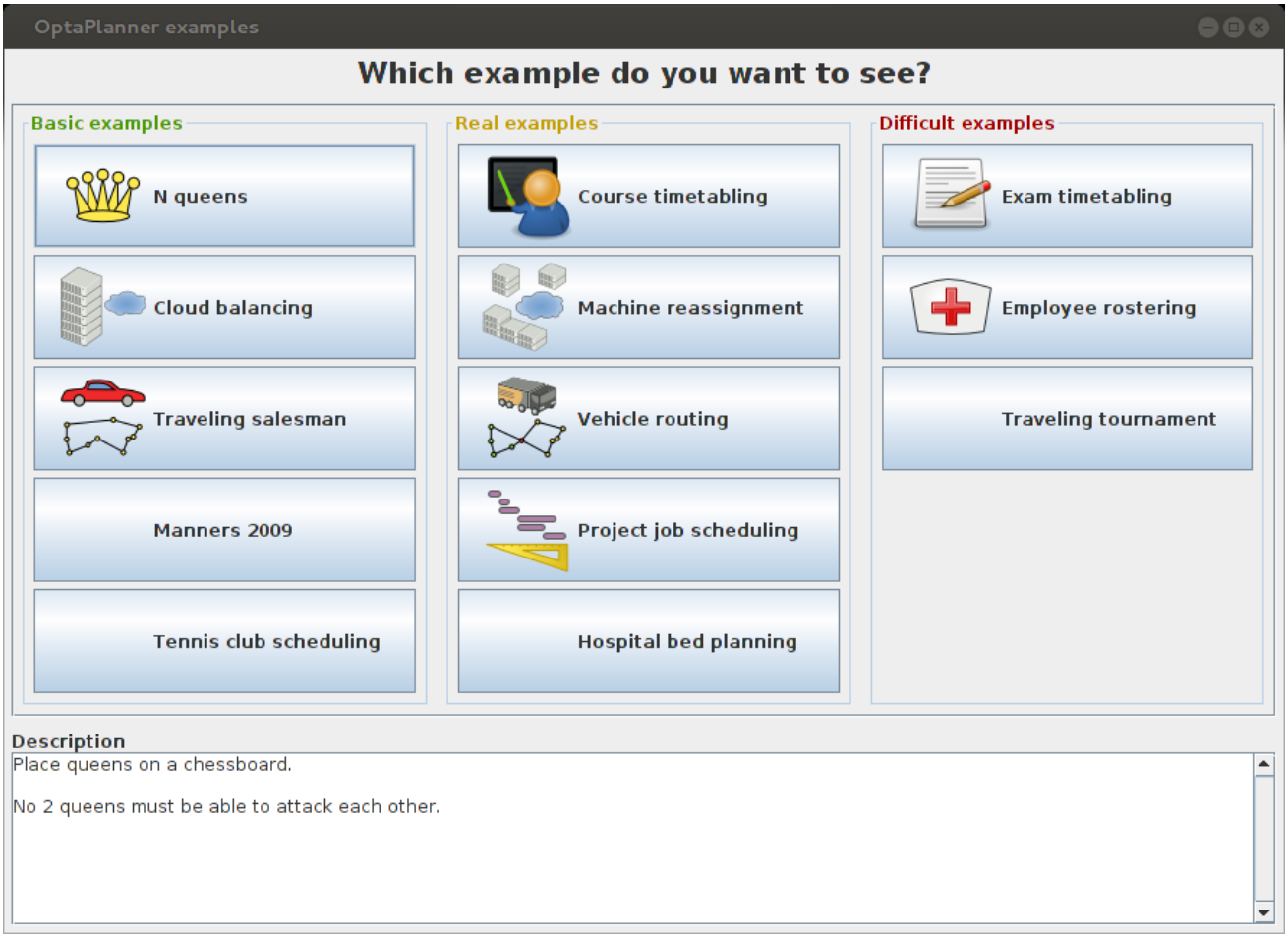
   Windows:

   ```
   $ cd examples
   $ runExamples.bat
   ```

図1.2 Business Resource Planner のダウンロード



サンプルの GUI アプリケーションが開きます。サンプルを 1 つ選択してください。



注記

Business Resource Planner 自体には、GUI の依存関係はありません。サーバーやモバイルの JVM で実行しても、表示される内容はデスクトップと同じです。

1.3. サンプルの実行

手順: 任意の **IDE** でサンプルを実行する方法

1. IDE を設定します。
   - IntelliJ および NetBeans の場合は、**examples/sources/pom.xml** を新規プロジェクトとして開けば、Maven の統合がその他の処理を行います。
   - Eclipse の場合は、**examples/sources** ディレクトリーで新規プロジェクトを開きます。
   - **binaries** ディレクトリーおよび **examples/binaries** ディレクトリーからクラスパスに、**examples/binaries/optaplanner-examples-*.jar** 以外のすべての JAR を追加します。
   - Java のソースディレクトリー **src/main/java** と Java のリソースディレクトリー **src/main/resources** を追加します。
2. 次に、実行を作成します。
   - 主要なクラス: **org.optaplanner.examples.app.OptaPlannerExamplesApp**
   - VM パラメーター (任意): **-Xmx512M -server**

- ワーキングディレクトリー: **examples** (このディレクトリーに **data** ディレクトリーが含まれます)

3. 実行設定を実行します。

手順: **Maven**、**Gradle**、**Ivy**、**Buildr**、または **Ant** での **Business Resource Planner** の使用

1. セントラルリポジトリー (および JBoss Nexus リポジトリー) から Business Resource Planner JAR を取得します。

2. Maven を使用する場合は、プロジェクトの **pom.xml** の **optaplanner-core** に依存関係を追加します。

```
<dependency>
  <groupId>org.optaplanner</groupId>
  <artifactId>optaplanner-core</artifactId>
  <version>...</version>
</dependency>
```

最新版については、セントラルリポジトリー を確認してください。これは、Gradle、Ivy および Buildr の場合も同様です。

3. (Ivy を使用せずに) Ant を使用する場合には、ダウンロードした Zip の **binaries** ディレクトリーからすべての JAR をコピーして、手作業でクラスパスに JAR が重複していないかを確認してください。

注記

ダウンロードした ZIP の **binaries** ディレクトリーには、**optaplanner-core** が使用しない JAR も数多く含まれます。**optaplanner-benchmark** など、他のモジュールが使用する JAR も含まれます。

Maven リポジトリー **pom.xml** ファイルで、特定のモジュールの特定のバージョンで最低限必要な依存関係を確認します。

注記

Examples are not a part of supported distribution. Customers can use them for learning purposes but Red Hat does not provide support for example code.

注記

以降、本書では Business Resource Planner を Planner と記載します。

第2章 クイックスタート

2.1. クラウドのバランスのチュートリアル

2.1.1. 問題の詳細

クラウドコンピューターを複数台所有し、そのクラウドコンピューターで複数のプロセスを実行する必要があると仮定します。以下の 4 つの制約がある中で、コンピューターに各プロセスを割り当てます。

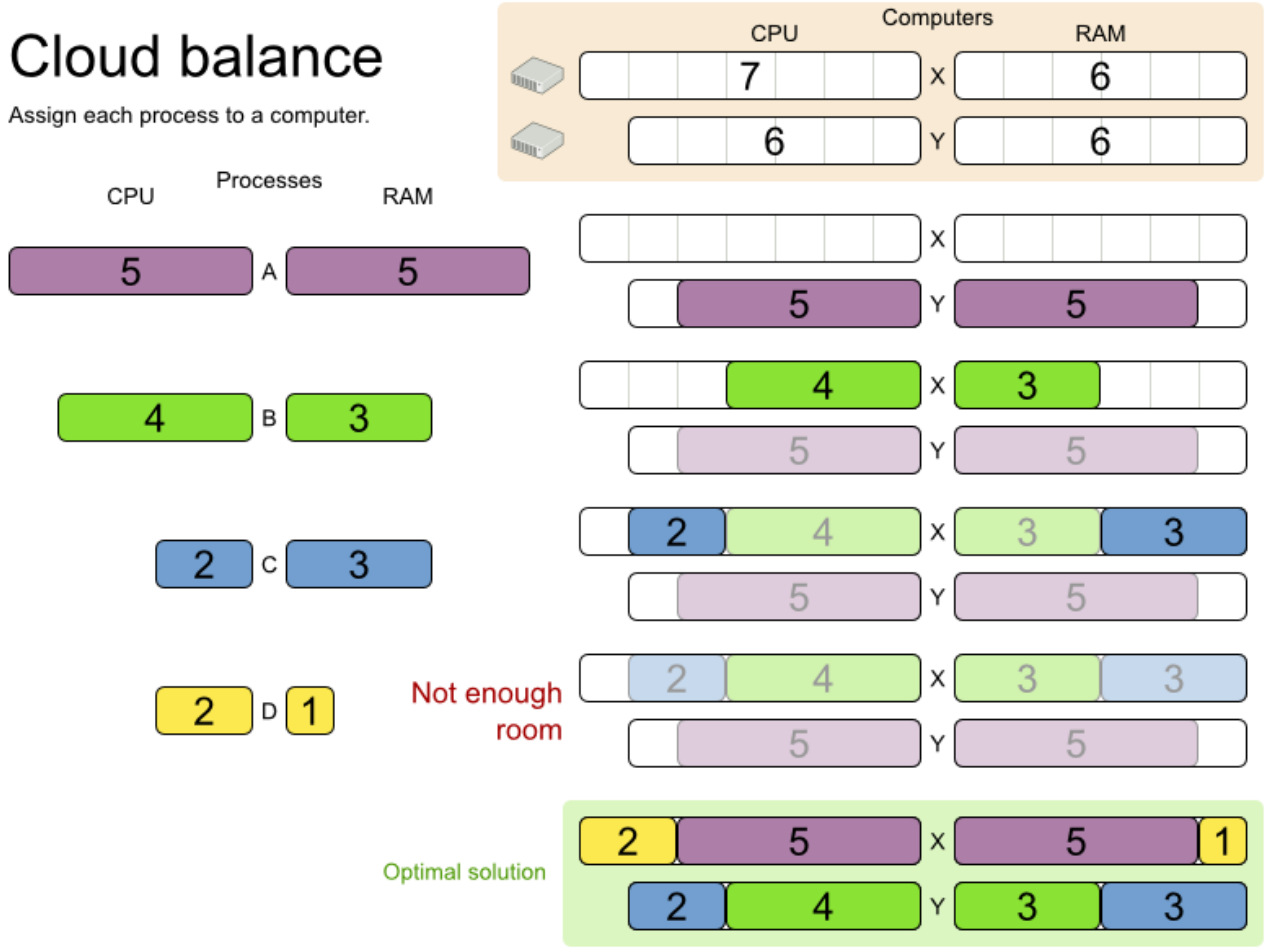
以下のハード制約を満たす必要があります。

- すべてのコンピューターで、プロセスの合計容量を処理するのに必要なハードウェア最小要件を満たす必要があります。
  - コンピューターには、最低でも、そのコンピューターに割り当てられたプロセスで必要とされる合計の CPU 処理能力が必要です。
  - コンピューターには、最低でも、そのコンピューターに割り当てられたプロセスで必要とされる合計のメモリーが必要です。
  - コンピューターには、最低でも、そのコンピューターに割り当てられたプロセスで必要とされる合計のネットワーク帯域幅が必要です。

以下のソフト制約を最適化する必要があります。

- 1 つまたは複数のプロセスが割り当てられたコンピューターにはそれぞれ、保守コストが発生します (コンピューターごとに固定)。
  - 合計保守コストを最小限に抑えます。

これは、ビンパッキング 問題にあたります。以下に、簡単なアルゴリズムを使用して、制約が 2 つ (CPU およびメモリー) があるコンピューター 2 台に、4 つのプロセスを割り当てるという簡単な例を紹介します。



ここで使用しているアルゴリズムは **FFD (First Fit Decreasing)** アルゴリズムです。このアルゴリズムでは、最初に大きいプロセスを割り当ててから、残りのスペースに小さいプロセスを割り当てていきます。図からも分かるように、黄色いプロセス「D」を割り当てる容量が残っていないため、最適ではありません。

Planner は、より適切な、別のアルゴリズムを使用して、より適した解 (ソリューション) を見つけます。また、データ (プロセス、コンピューター) と制約 (ハードウェア要件やその他の制約) の両方を増やして評価します。では、以下のシナリオで Planner をどのように使用できるかをご覧ください。

2.1.2. 問題の規模

表2.1 クラウドのバランス問題の規模

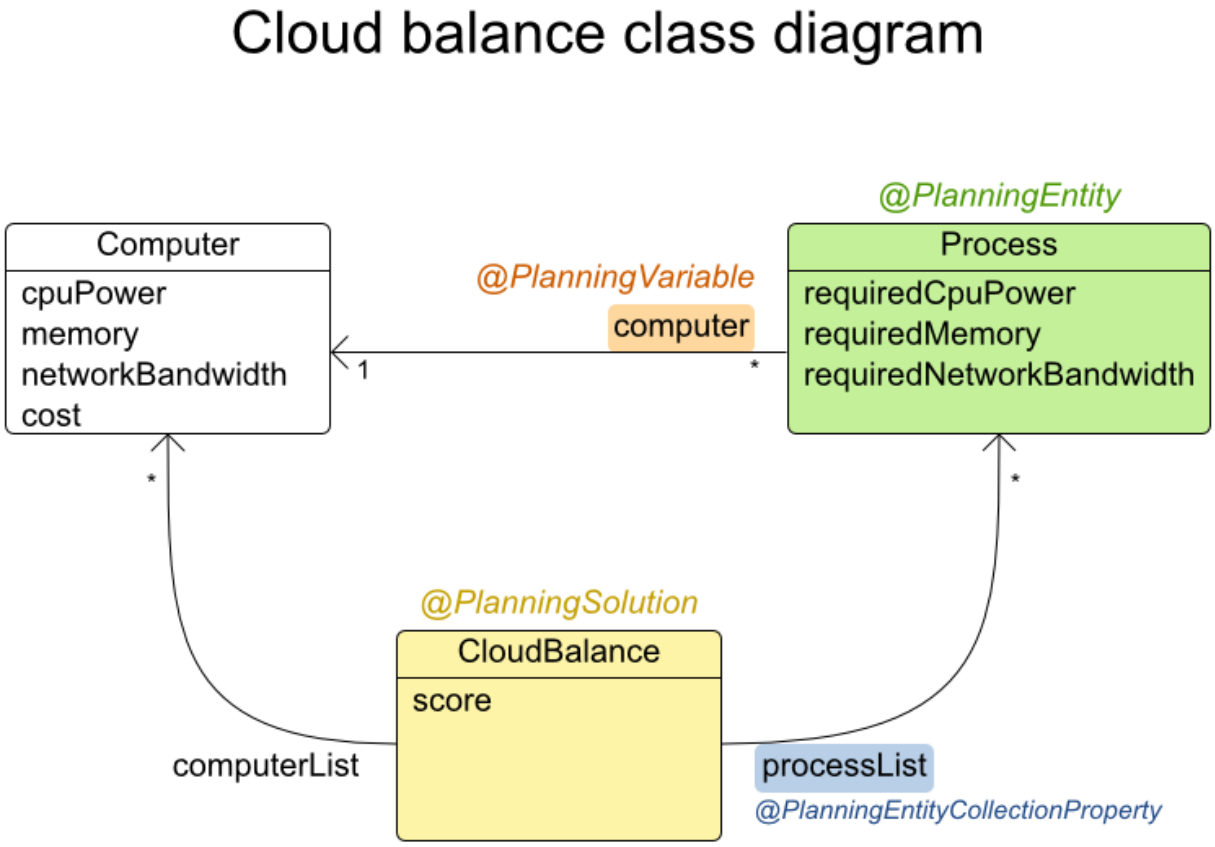
問題の規模	コンピューター	プロセス	探索空間
コンピューター 2 台、プロセス 6 件	2	6	64
コンピューター 3 台、プロセス 9 件	3	9	10^4
コンピューター 4 台、プロセス 12 件	4	12	10^7

問題の規模	コンピューター	プロセス	探索空間
コンピューター 100 台、プロセス 300 件	100	300	10^{600}
コンピューター 200 台、プロセス 600 件	200	600	10^{1380}
コンピューター 400 台、プロセス 1200 件	400	1200	10^{3122}
コンピューター 800 台、プロセス 2400 件	800	2400	10^{6967}

2.1.3. ドメインモデルの設計

ドメインモデルから始めましょう。

- **Computer**: 特定のハードウェア (CPU 処理能力、RAM メモリー、ネットワーク帯域幅) が搭載され、特定の保守コストが発生するコンピューターを表します。
- **Process**: デマンドのあるプロセスを表します。このプロセスは、Planner によって **Computer** に割り当てられます。
- **CloudBalance**: 問題を表します。CloudBalance には、特定のデータセットに関する **Computer** および **Process** がすべて含まれます。



上の UML クラスの図では、Planner のコンセプトにすでにアノテーションが付けてあります。

- PlanningEntity (プランニングエンティティー): 計画時に変化するクラス。この例では、**Process** クラスがそれにあたります。
- PlanningVariable (プランニング変数): 計画時に変化するプランニングエンティティークラスのプロパティー。この例では、**Process** クラスの **computer** プロパティーがそれにあたります。
- PlanningSolution (ソリューション): プランニングエンティティーをすべて含むデータセットを表現するクラス。この例では、**CloudBalance** クラスがそれにあたります。

2.1.4. メインメソッド

では、実際に試してみましょう。任意の IDE にサンプルを ダウンロードして、設定します。**org.optaplanner.examples.cloudbalancing.app.CloudBalancingHelloWorld** を実行します。デフォルトでは、120 秒間実行するように設定されています。これにより、以下のコードが実行します。

例2.1 CloudBalancingHelloWorld.java

```
public class CloudBalancingHelloWorld {

    public static void main(String[] args) {
        // Build the Solver
        SolverFactory<CloudBalance> solverFactory = SolverFactory.createFromXmlResource(
                "org/optaplanner/examples/cloudbalancing/solver/cloudBalancingSolverConfig.xml");
        Solver<CloudBalance> solver = solverFactory.buildSolver();
```

```
        // Load a problem with 400 computers and 1200 processes
        CloudBalance unsolvedCloudBalance = new
CloudBalancingGenerator().createCloudBalance(400, 1200);

        // Solve the problem
        CloudBalance solvedCloudBalance = solver.solve(unsolvedCloudBalance);

        // Display the result
        System.out.println("\nSolved cloudBalance with 400 computers and 1200 processes:\n"
                + toDisplayString(solvedCloudBalance));
    }

    ...
}
```

このコードサンプルにより、以下が行われます。

- Solver の設定をもとに **Solver** を構築します (ここでは、クラスパスの XML ファイル を使用します)。

```
        SolverFactory<CloudBalance> solverFactory = SolverFactory.createFromXmlResource(

"org/optaplanner/examples/cloudbalancing/solver/cloudBalancingSolverConfig.xml");
        Solver solver<CloudBalance> = solverFactory.buildSolver();
```

- 問題を読み込みます。**CloudBalancingGenerator** が無作為に問題を生成します。これを、たとえばデータベースから実際の問題を読み込むクラスに置き換えてください。

```
        CloudBalance unsolvedCloudBalance = new
CloudBalancingGenerator().createCloudBalance(400, 1200);
```

- 問題を解決します。

```
        CloudBalance solvedCloudBalance = solver.solve(unsolvedCloudBalance);
```

- 結果を表示します。

```
        System.out.println("\nSolved cloudBalance with 400 computers and 1200 processes:\n"
                + toDisplayString(solvedCloudBalance));
```

Solver の構築部分だけが唯一複雑になります。これについては、次のセクションで詳しく説明します。

2.1.5. Solver の設定

Solver の設定を見てみましょう。

例2.2 **cloudBalancingSolverConfig.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
  <!-- Domain model configuration -->
```

```
  <scanAnnotatedClasses/>

  <!-- Score configuration -->
  <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

  <easyScoreCalculatorClass>org.optaplanner.examples.cloudbalancing.solver.score.CloudBalancing
  EasyScoreCalculator</easyScoreCalculatorClass>
     <!--
  <scoreDrl>org/optaplanner/examples/cloudbalancing/solver/cloudBalancingScoreRules.drl</scoreDr
  >-->
  </scoreDirectorFactory>

  <!-- Optimization algorithms configuration -->
  <termination>
    <secondsSpentLimit>30</secondsSpentLimit>
  </termination>
</solver>
```

Solver の設定は、3 つの部分で構成されます。

- ドメインモデルの設定: Planner が変更できるものは何ですか? Planner にドメインクラスを指定する必要があります。ここでは、(**@PlanningEntity** または **@PlanningSolution** アノテーションに対して) クラスパス内の全クラスを自動的にスキャンします。

  ```
  <scanAnnotatedClasses/>
  ```

- スコアの設定: Planner はプランニング変数をどのように最適化すればよいでしょうか? また、目的は何でしょうか? ここでは、ハード制約とソフト制約を使用するため、**HardSoftScore** を使用します。ただし、Planner に、ビジネス要件に合ったスコアの計算方法を指定する必要があります。後ほど、スコアの計算方法を 2 種類 (単純な Java 実装の使用、または Drools DRL の使用) 紹介します。

  ```
  <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

  <easyScoreCalculatorClass>org.optaplanner.examples.cloudbalancing.solver.score.CloudBala
  ncingEasyScoreCalculator</easyScoreCalculatorClass>
     <!--
  <scoreDrl>org/optaplanner/examples/cloudbalancing/solver/cloudBalancingScoreRules.drl</scc
  eDrl>-->
  </scoreDirectorFactory>
  ```

- 最適化のアルゴリズム設定: Planner はどのように最適化しますか? この例では、(最適化アルゴリズムが明示的に設定されていないので) 30 秒間、デフォルトの 最適化アルゴリズム を使用します。

  ```
  <termination>
    <secondsSpentLimit>30</secondsSpentLimit>
  </termination>
  ```

  Planner は、数秒 (リアルタイム計画 では 15 ミリ秒未満になる場合も) でも良い結果は得られますが、時間が長くなればなるほど、結果は良くなります。高度なユースケースでは、ハードな時間制限以外に、終了の条件 を使用することが適しています。

デフォルトのアルゴリズムでも、人的作業による計画や、多くの社内の実装よりもはるかに優れていますが、さらに良くするためには、ベンチマーカー を使用して 高度な調整 (Power tweak) を行います。

では、ドメインモデルクラスとスコア設定について検証しましょう。

2.1.6. ドメインモデルの実装

2.1.6.1. Computer クラス

Computer クラスは POJO (Plain Old Java Object) で、一般的には、この種類のクラスがよく使用されます。

例2.3 CloudComputer.java

```
public class CloudComputer ... {

    private int cpuPower;
    private int memory;
    private int networkBandwidth;
    private int cost;

    ... // getters
}
```

2.1.6.2. Process クラス

Process クラスは、特に重要です。このクラスが **computer** フィールドを変更できることを Planner に指定する必要があるため、このクラスに **@PlanningEntity**、**getComputer()** ゲッターに **@PlanningVariable** アノテーションを付けます。

例2.4 CloudProcess.java

```
@PlanningEntity(...)
public class CloudProcess ... {

    private int requiredCpuPower;
    private int requiredMemory;
    private int requiredNetworkBandwidth;

    private CloudComputer computer;

    ... // getters

    @PlanningVariable(valueRangeProviderRefs = {"computerRange"})
    public CloudComputer getComputer() {
        return computer;
    }

    public void setComputer(CloudComputer computer) {
        computer = computer;
    }
```

```
    // ************************************************************************
    // Complex methods
    // ************************************************************************

    ...

}
```

Planner が **computer** フィールドに選択できる値は、**Solution** 実装の **CloudBalance.getComputerList()** メソッドから取得します。このメソッドは、現在のデータセット内のコンピューター一覧を返します。**valueRangeProviderRefs**プロパティーを使用して、この情報を Planner に渡します。

注記

ゲッターアノテーションの代わりに、フィールドアノテーション を使用することもできます。

2.1.6.3. CloudBalance クラス

CloudBalance クラスは、**Solution** インターフェースを実装します。これは、コンピューターおよびプロセスの一覧を保持します。変更可能なプロセスのコレクションを取得する方法を Planner に指定する必要があるため、**getProcessList** のゲッターに **@PlanningEntityCollectionProperty** アノテーションを付ける必要があります。

CloudBalance クラスには、**score** プロパティーも含まれます。このプロパティーは、対象 **Solution** インスタンスにおけるそのときの スコア を表します。

例2.5 CloudBalance.java

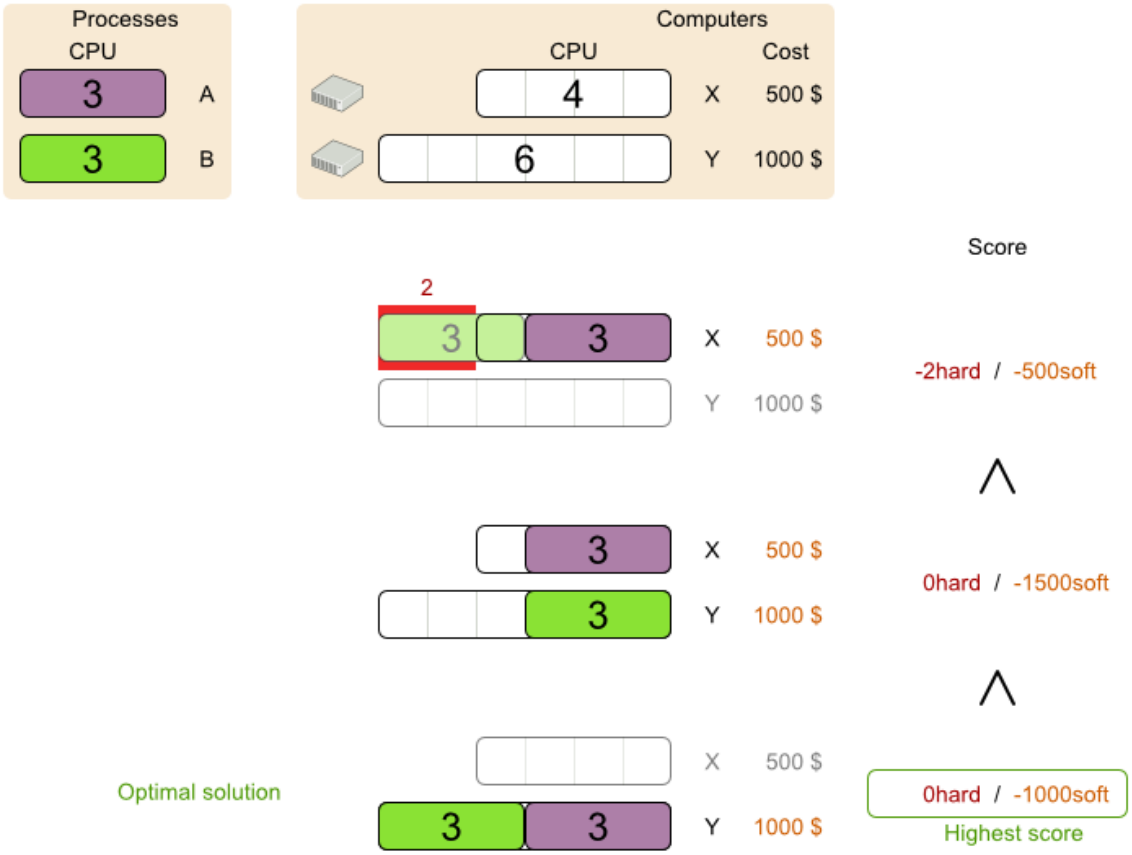
```
@PlanningSolution
public class CloudBalance ... implements Solution<HardSoftScore> {

    private List<CloudComputer> computerList;

    private List<CloudProcess> processList;

    private HardSoftScore score;

    @ValueRangeProvider(id = "computerRange")
    public List<CloudComputer> getComputerList() {
        return computerList;
    }

    @PlanningEntityCollectionProperty
    public List<CloudProcess> getProcessList() {
        return processList;
    }

    ...

    public HardSoftScore getScore() {
        return score;
    }
```

```
    public void setScore(HardSoftScore score) {
        this.score = score;
    }

    // ************************************************************************
    // Complex methods
    // ************************************************************************

    public Collection<? extends Object> getProblemFacts() {
        List<Object> facts = new ArrayList<Object>();
        facts.addAll(computerList);
        // Do not add the planning entity's (processList) because that will be done automatically
        return facts;
    }

    ...
}
```

getProblemFacts() メソッドは、スコア計算に Drools を使用した場合に必要になります。他のスコア計算タイプには必要ありません。

2.1.7. スコアの設定

Planner は、スコアが最も高いソリューションを探します。この例では、**HardSoftScore** を使用し、Planner がハード制約に違反していない (またはハードウェア要件を満たす) ソリューションと、ソフト制約に違反する数が少ない (メンテナンスコストが低い) ソリューションを探します。



また、Planner には、ドメイン固有のスコア制約についても指定する必要があります。このようなスコア関数の実装には複数の方法があります。

- Easy Java
- Java インクリメント演算子
- Drools

2 つの異なる実装について見ていきます。

2.1.7.1. Easy Java のスコア設定

スコア関数の定義方法の 1 つに、プレーン Java での **EasyScoreCalculator** インターフェース実装があります。

```
  <scoreDirectorFactory>
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

  <easyScoreCalculatorClass>org.optaplanner.examples.cloudbalancing.solver.score.CloudBalancingEas
  yScoreCalculator</easyScoreCalculatorClass>
    </scoreDirectorFactory>
```

calculateScore(Solution) メソッドを実装すれば **HardSoftScore** インスタンスを返します。

例2.6 CloudBalancingEasyScoreCalculator.java

```
public class CloudBalancingEasyScoreCalculator implements
EasyScoreCalculator<CloudBalance> {

    /**
     * A very simple implementation. The double loop can easily be removed by using Maps as
shown in
     * {@link CloudBalancingMapBasedEasyScoreCalculator#calculateScore(CloudBalance)}.
     */
    public HardSoftScore calculateScore(CloudBalance cloudBalance) {
        int hardScore = 0;
        int softScore = 0;
        for (CloudComputer computer : cloudBalance.getComputerList()) {
            int cpuPowerUsage = 0;
            int memoryUsage = 0;
            int networkBandwidthUsage = 0;
            boolean used = false;

            // Calculate usage
            for (CloudProcess process : cloudBalance.getProcessList()) {
                if (computer.equals(process.getComputer())) {
                    cpuPowerUsage += process.getRequiredCpuPower();
                    memoryUsage += process.getRequiredMemory();
                    networkBandwidthUsage += process.getRequiredNetworkBandwidth();
                    used = true;
                }
            }

            // Hard constraints
            int cpuPowerAvailable = computer.getCpuPower() - cpuPowerUsage;
            if (cpuPowerAvailable < 0) {
                hardScore += cpuPowerAvailable;
            }
            int memoryAvailable = computer.getMemory() - memoryUsage;
            if (memoryAvailable < 0) {
                hardScore += memoryAvailable;
            }
            int networkBandwidthAvailable = computer.getNetworkBandwidth() -
networkBandwidthUsage;
            if (networkBandwidthAvailable < 0) {
                hardScore += networkBandwidthAvailable;
            }

            // Soft constraints
            if (used) {
                softScore -= computer.getCost();
            }
        }
        return HardSoftScore.valueOf(hardScore, softScore);
    }

}
```

Maps を使用して最適化し、**processList** を 1 回だけ反復した場合でも、インクリメンタルスコアの計算 が行われないため、処理が遅くなります。これを修正するには、Java インクリメント演算子のスコ

ア関数、または Drools のスコア関数のいずれかを使用します。では、Drools スコア関数について見ていきます。

2.1.7.2. Drools スコア関数

クラスパスに **scoreDrl** リソースを追加すれば、Drools ルールエンジンをスコア関数として使用できます。

```
  <scoreDirectorFactory>
   <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>

<scoreDrl>org/optaplanner/examples/cloudbalancing/solver/cloudBalancingScoreRules.drl</scoreDrl
>
  </scoreDirectorFactory>
```

まず、すべてのコンピューターに、すべてのプロセスに対応するのに十分な CPU、メモリー、ネットワーク帯域幅があることを確認します。これをハード制約とします。

例2.7 **cloudBalancingScoreRules.drl** – ハード制約

```
...

import org.optaplanner.examples.cloudbalancing.domain.CloudBalance;
import org.optaplanner.examples.cloudbalancing.domain.CloudComputer;
import org.optaplanner.examples.cloudbalancing.domain.CloudProcess;

global HardSoftScoreHolder scoreHolder;

// ############################################################################
// Hard constraints
// ############################################################################

rule "requiredCpuPowerTotal"
    when
        $computer : CloudComputer($cpuPower : cpuPower)
        $requiredCpuPowerTotal : Number(intValue > $cpuPower) from accumulate(
            CloudProcess(
                computer == $computer,
                $requiredCpuPower : requiredCpuPower),
            sum($requiredCpuPower)
        )
    then
        scoreHolder.addHardConstraintMatch(kcontext, $cpuPower -
$requiredCpuPowerTotal.intValue());
end

rule "requiredMemoryTotal"
    ...
end

rule "requiredNetworkBandwidthTotal"
    ...
end
```

次に、これらの制約を満たした場合に、メンテナンスコストを最小限に抑えられるように、ソフト制約としてその条件を追加します。

例2.8 cloudBalancingScoreRules.drl - ソフト制約

```
// ############################################################################
// Soft constraints
// ############################################################################

rule "computerCost"
    when
        $computer : CloudComputer($cost : cost)
        exists CloudProcess(computer == $computer)
    then
        scoreHolder.addSoftConstraintMatch(kcontext, - $cost);
end
```

スコアの計算に Drools ルールエンジンを使用する場合は、デシジョンテーブル (エクセルまたは Web ベース) や KIE Workbench など、他の Drools 技術と統合することができます。

2.1.8. このチュートリアルの応用

上記のようなシンプルな例が設定できたら、さらに掘り下げてみてください。ドメインモデルを改良して、以下のような制約を追加してみてください。

- すべての プロセス が サービス に属する。コンピューターはクラッシュする可能性があるため、同じサービスを実行するプロセスは、別のコンピューターに割り当てる必要がある。
- すべてのコンピューターがビルに設置されている。ビルが火災にあう可能性があるので、同じサービスのプロセスは、別のビルに設置されているコンピューターに割り当てる必要がある。

第3章 ユースケースおよびサンプル

3.1. サンプルの概要

Planner にはサンプルが複数含まれます。本書では、N クィーンの例とクラウドのバランスの例を主に使用するため、少なくてもこの 2 つのセクションは目を通しておいてください。

サンプルのソースコードは、**examples/sources** に保存されているディストリビューション ZIP か、**optaplanner/optaplanner-examples** の git から入手できます。

表3.1 サンプルの概要

例	ドメイン	サイズ	コンペティションの有無	使用する特別機能
N クィーン	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 256 • 値 ⇐ 256 • 探索空間 ⇐ 10^{616}	• 意味がない (不正が可能)	なし
クラウドのバランス	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 2400 • 値 ⇐ 800 • 探索空間 ⇐ 10^{6967}	• なし • 弊社が定義	• リアルタイム計画
巡回セールスマン	• エンティティークラス 1 つ • 連鎖変数 1 つ	• エンティティー ⇐ 980 • 値 ⇐ 980 • 探索空間 ⇐ 10^{2927}	• 現実的でない • TSP web	• リアルタイム計画
ディナーパーティー	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 144 • 値 ⇐ 72 • 探索空間 ⇐ 10^{310}	• 現実的でない	• スコア制約のデシジョンテーブルのスプレッドシート (XLS)

例	ドメイン	サイズ	コンペティションの有無	使用する特別機能
テニスクラブのスケジュール	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 72 • 値 ⇐ 7 • 探索空間 ⇐ 10^{60}	• なし • 弊社が定義	• 公平性を保つためのスコア制約 • 移動できないエンティティー
会議のスケジュール	• エンティティークラス 1 つ • 変数 2 つ	• エンティティー ⇐ 10 • 値 ⇐ 320 および ⇐ 5 • 探索空間 ⇐ 10^{320}	• なし • 弊社が定義	• TimeGrain パターン
コースの時間割	• エンティティークラス 1 つ • 変数 2 つ	• エンティティー ⇐ 434 • 値 ⇐ 25 および ⇐ 20 • 探索空間 ⇐ 10^{1171}	• 現実的 • ITC 2007 track 3	• 移動できないエンティティー
マシンの再割当て	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 50000 • 値 ⇐ 5000 • 探索空間 ⇐ 10^{184948}	• **ほぼ現実的** • ROADEF 2012	• リアルタイム計画

例	ドメイン	サイズ	コンペティションの有無	使用する特別機能
配送経路	• エンティティークラス1つ • 連鎖変数1つ • シャドウエンティティークラス1つ • 自動シャドウ変数1つ	• エンティティー ⇐ 134 • 値 ⇐ 141 • 探索空間 ⇐ 10^{285}	• 現実的でない • VRP web	• シャドウ変数 • リアルタイム計画 • 近傍選択 • 実際の路上距離
時間枠がある中での 配送経路	配送経路に関する追加機能: • シャドウ変数1つ	• エンティティー ⇐ 1000 • 値 ⇐ 1250 • 探索空間 ⇐ 10^{3000}	• 現実的でない • VRP web	配送経路に関する追加機能: • カスタムの VariableListener
プロジェクトジョブのスケジュール	• エンティティークラス1つ • 変数2つ • シャドウ変数1つ	• エンティティー ⇐ 640 • 値 ⇐ ? および ⇐ ? • 探索空間 ⇐ ?	• ほぼ**現実的** • MISTA 2013	• 丸められるスコア • カスタムの VariableListener • ValueRangeFactory
病床計画	• エンティティークラス1つ • null 許容型変数1つ	• エンティティー ⇐ 2750 • 値 ⇐ 471 • 探索空間 ⇐ 10^{6851}	• 現実的でない • Kaho PAS	• 過制約計画

例	ドメイン	サイズ	コンペティションの有無	使用する特別機能
試験の時間割	• エンティティークラス 2 つ (同じ階層) • 変数 2 つ	• エンティティー ⇐ 1096 • 値 ⇐ 80 and ⇐ 49 • 探索空間 ⇐ 10^{3374}	• **現実的** • ITC 2007 track 1	• カスタムの VariableListener
従業員の勤務表	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 752 • 値 ⇐ 50 • 探索空間 ⇐ 10^{1277}	• **現実的** • INRC 2010	• 継続的計画 • リアルタイム計画
巡回トーナメント	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 1560 • 値 ⇐ 78 • 探索空間 ⇐ 10^{2951}	• 現実的でない • TTP	• カスタムの MoveListFactory
コストを抑えるスケジュール	• エンティティークラス 1 つ • 変数 2 つ	• エンティティー ⇐ 500 • 値 ⇐ 100 および ⇐ 288 • 探索空間 ⇐ 10^{20078}	• ほぼ**現実的** • ICON Energy	• フィールドアノテーション • ValueRangeFactory
投資	• エンティティークラス 1 つ • 変数 1 つ	• エンティティー ⇐ 11 • 値 = 1000 • 探索空間 ⇐ 10^{4}	• なし • 弊社が定義	• ValueRangeFactory

現実的なコンペティション とは、以下のような公式で独立したコンペティションを指します。

- 実際のユースケースを明確に定義するコンペティション
- 実際の制約があるコンペティション
- 実際のデータセットが複数あるコンペティション
- 特定のハードウェアで特定の時間内に結果を再現できるコンペティション
- 教育機関および/または企業の運用研究コミュニティーが真剣に参加しているコンペティション

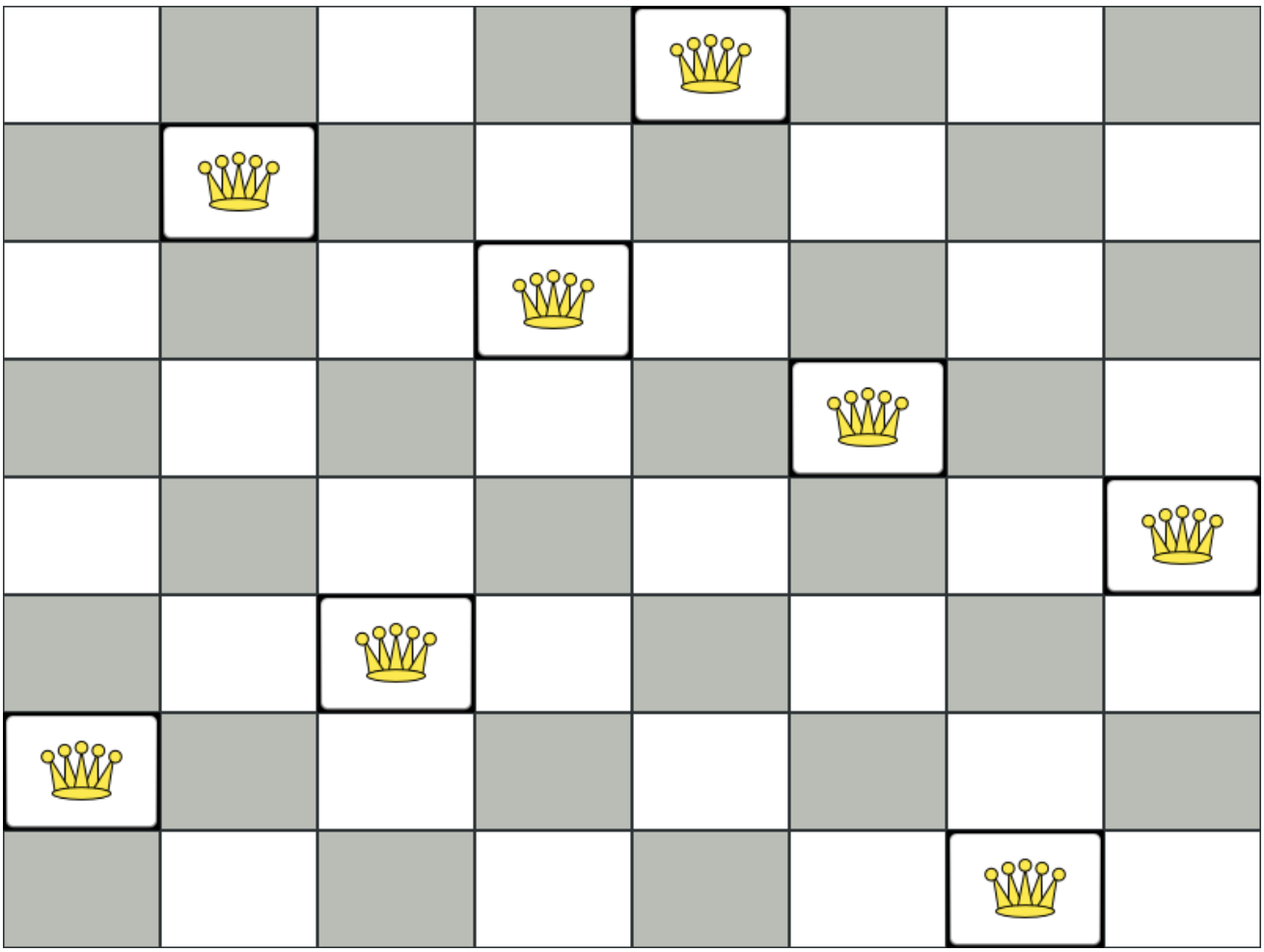
このような現実的なコンペティションでは、Planner を、競合のソフトウェアや教育研究と客観的に比較することができます。

3.2. 基本例

3.2.1. N クィーン

3.2.1.1. 問題の詳細

n サイズのチェスボードで、他のクィーンに取られない位置に n 個のクィーンを置きます。最も一般的なクィーンパズルは、**n = 8** の 8 個のクィーンパズルです。



制約:

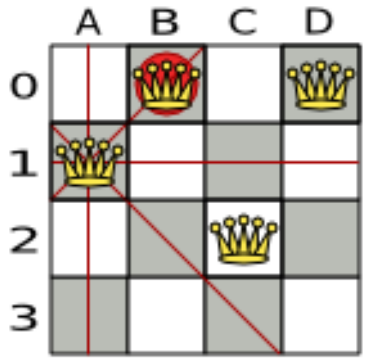
- **n** 行および **n** 列のチェスボードを使用します。

- チェスボードに n 個のクィーンを置きます。
- クィーンが他のクィーンに取られないように配置します。クィーンは、同じ水平線上、垂直線上、対角線上にある他のクィーンを取ることができます。

本書の例では、クィーン 4 個のパズルを主に使用します。

以下が提案された解です。

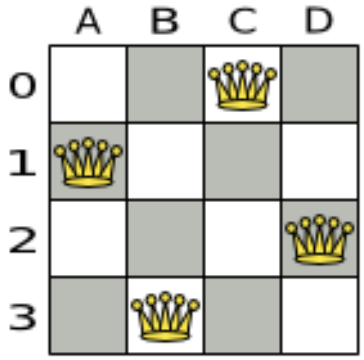
図3.1 クィーン 4 個のパズルの誤った解



上記の解は、**A1** と **B0** (および **B0** と **D0**) のクィーンがお互いに駒を取れるので間違っています。**B0** のクィーンをどかせば「他のクィーンに取られないようにする」という制約は順守できますが、「n 個のクィーンを置く」という制約に違反します。

以下は正しい解です。

図3.2 クィーン 4 個のパズルの正しい解



すべての制約が満たされているので、これが正解です。n クィーンパズルでは、正解が複数存在する場合が多々あります。各 n に対して、考えられるすべての解を見つけるのではなく、正しい解を 1 つ導き出すことにフォーカスします。

3.2.1.2. 問題の規模

```
4queens   has   4 queens with a search space of    256.
8queens   has   8 queens with a search space of   10^7.
16queens  has  16 queens with a search space of  10^19.
32queens  has  32 queens with a search space of  10^48.
64queens  has  64 queens with a search space of 10^115.
256queens has 256 queens with a search space of 10^616.
```

N クィーンは、初心者用のサンプルとして実装されているため、最適化はされていませんが、クィーンが 64 個になっても簡単に処理できます。何点か変更を加えると、クィーンが 5000 個以上になっても簡単に対応できることが立証されています。

3.2.1.3. ドメインモデル

適切なドメインモデルを使用してください。計画の問題の理解と解決がより簡単になります。以下は、n クィーン問題を例にドメインモデルを使用したものです。

```
public class Column {

    private int index;

    // ... getters and setters
}
```

```
public class Row {

    private int index;

    // ... getters and setters
}
```

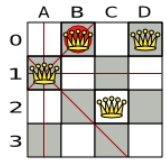
```
public class Queen {

    private Column column;
    private Row row;

    public int getAscendingDiagonalIndex() {...}
    public int getDescendingDiagonalIndex() {...}

    // ... getters and setters
}
```

Queen インスタンスには **Column** (例: 0 は列 A、1 は列 B) および **Row** (例: 0 は行 0、1 は行 1) が含まれます。列と行をもとに、昇順の対角線、降順の対角線を計算することができます。列と行のインデックスは、チェスボードの左上隅から数えています。

```
public class NQueens implements Solution<SimpleScore> {

    private int n;
    private List<Column> columnList;
    private List<Row> rowList;

    private List<Queen> queenList;

    private SimpleScore score;

    // ... getters and setters
}
```

1 つの **NQueens** インスタンスには **Queen** インスタンスの一覧が含まれています。これが **Solution** 実装として提供され、Solver が解決して読み出します。たとえば、4 クイーンの例では、NQueens の **getN()** メソッドが常に 4 を返します。

表3.2 ドメインモデルでの 4 クィーンの解

解	クィーン	columnIndex	rowIndex	ascendingDiagonalIndex (columnIndex + rowIndex)	descendingDiagonalIndex (columnIndex - rowIndex)
	A1	0	1	1 (**)	-1
	B0	1	0 (*)	1 (**)	1
	C2	2	2	4	0
	D0	3	0 (*)	3	3

(*) や (**) のように、クィーン 2 つが同じ行、列、対角線を共有する場合は、2 つの駒が互いを取ることができます。

3.2.2. クラウドのバランス

この例は、チュートリアル で説明されています。

3.2.3. 巡回セールスマン (TSP - 巡回セールスマン問題)

3.2.3.1. 問題の詳細

都市の一覧をもとに、セールスマンが最短距離で、各都市を 1 度だけ訪問するルートを探します。

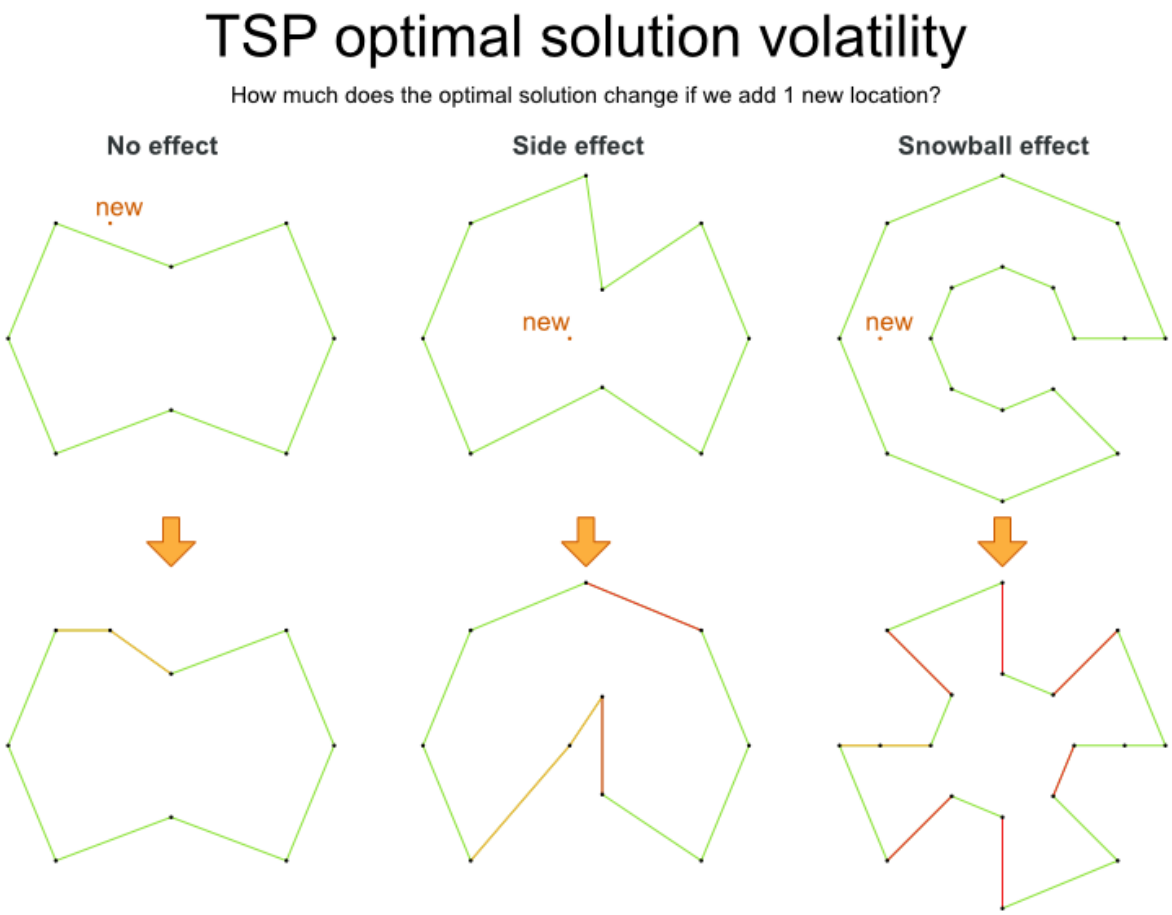
この問題は ウィキペディア に定義されています。これは、計算数学で 最も熱心に研究された問題の 1 つです。大概は、従業員のシフト勤務など、その他の制約と一緒に計画の問題の一部として使用されます。

3.2.3.2. 問題の規模

```
dj38     has  38 cities with a search space of   10^58.
europe40 has  40 cities with a search space of   10^62.
st70     has  70 cities with a search space of  10^126.
pcb442   has 442 cities with a search space of 10^1166.
lu980    has 980 cities with a search space of 10^2927.
```

3.2.3.3. 問題の難易度

TSP の定義は単純ですが、問題の解決は驚くほど難しくなります。これは NP 困難問題と呼ばれ、多くの計画の問題と同様、特定の問題のデータセットに対する最適な解は、その問題のデータセットが少しでも変更すると、大幅に変化する可能性があります。



3.2.4. ディナーパーティー

3.2.4.1. 問題の詳細

マナーズさんがディナーパーティを再度開催することにしました。

- 今回は、ゲスト 144 名を招待し、円卓を 12 個用意し、各テーブルに椅子を 12 席用意しました。
- 座席は、同じ性別の人が隣同士にならないようにします。
- また、隣の人とは必ず、同じ趣味を 1 つ持つようにします。
- 各テーブルには、政治家、医者、著名人、コーチ、教師、そしてプログラマーを 2 名ずつ配置します。
- 政治家、医者、コーチ、プログラマーは、それぞれ同じ業種の人が同じテーブルにならないようにします。

Drools Expert にも、(サイズがはるかに小さい) 一般的なミスマナーズの例が含まれており、包括的なヒューリスティックを採用して解を導き出しています。Planner の実装は、ヒューリスティックを使用して最適解を見つけ、Drools Expert を使用して各解のスコアを計算するため、スケーラビリティーがはるかに高くなっています。

3.2.4.2. 問題の規模

> wedding01 has 18 jobs, 144 guests, 288 hobby practicians, 12 tables and 144 seats with a search space of 10^310.

3.2.5. テニスクラブのスケジュール

3.2.5.1. 問題の詳細

テニスクラブでは、毎週 4 チームが総あたりで試合をします。4 つの対戦枠を公平にチームに割り当てます。

ハード制約:

- 制約: チームは 1 日に 1 回だけ試合ができる。
- 参加不可: 日程によって参加できないチームがある。

中程度の制約:

- 公平な割り当て: 各チームが試合をする回数を (ほぼ) 同じにする。

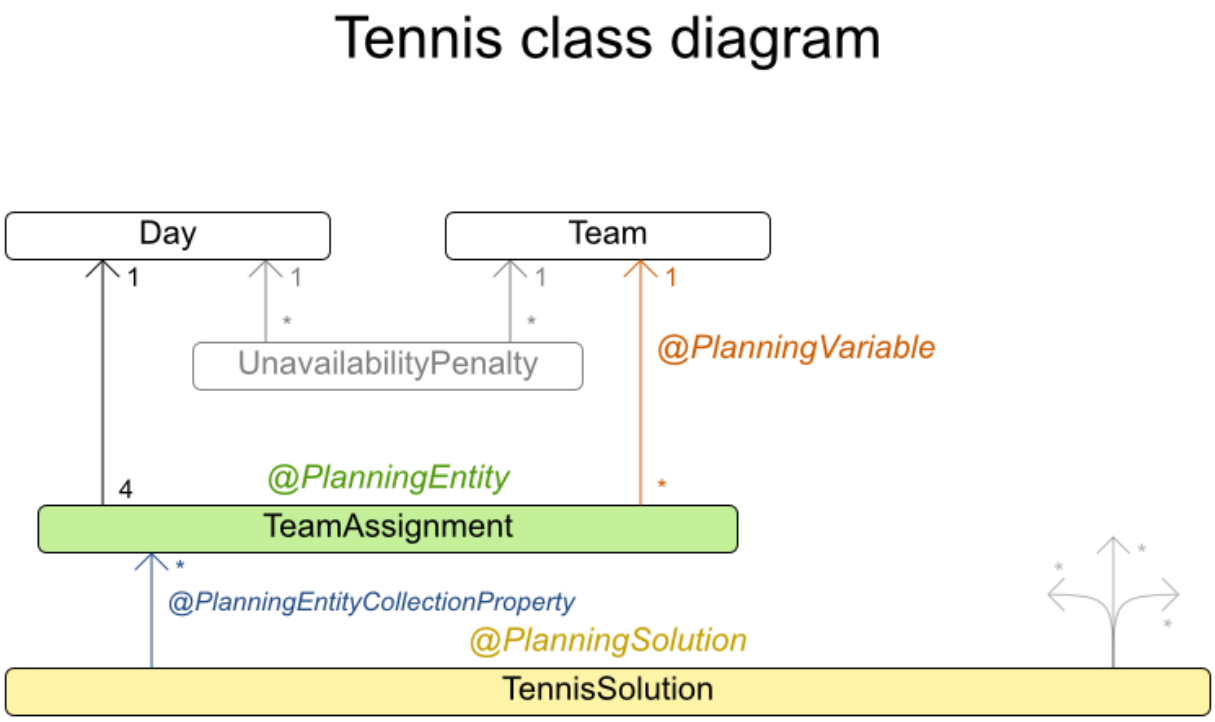
ソフト制約:

- 均等に対戦: 各チームが、各対戦相手と対戦する回数を同じにする。

3.2.5.2. 問題の規模

> munich-7teams has 7 teams, 18 days, 12 unavailabilityPenalties and 72 teamAssignments with a search space of 10^60.

3.2.5.3. ドメインモデル



3.2.6. 会議のスケジュール

3.2.6.1. 問題の詳細

各会議に、開始時間と会議室を割り当てます。会議の長さは異なります。

ハード制約:

- 部屋の制約: 2 つの会議が、同じ時間に同じ会議室を使用することはできない。
- 必須の出席者: 同じ時間に開催される必須の会議を 2 つ割り当てることはできない。

中程度の制約:

- 任意の出席者: 同じ時間に開催される任意の会議を 2 つ割り当てることはできない。また、任意の会議、および必須の会議をそれぞれ 1 つ割り当てることはできない。

ソフト制約:

- 早い段階でスケジュール: すべての会議をできるだけ早くスケジュールする。

3.2.6.2. 問題の規模

> 50meetings-160timegrains-5rooms has 50 meetings, 160 timeGrains and 5 rooms with a search space of 10^145.
> 100meetings-320timegrains-5rooms has 100 meetings, 320 timeGrains and 5 rooms with a search space of 10^320.

-

3.3. 実例

3.3.1. コースの時間割 (ITC 2007 Track 3 - カリキュラムのスケジュール)

3.3.1.1. 問題の詳細

各授業を、時間枠および講義室に割り当ててスケジュールを組みます。

ハード制約:

- 講師の制約: 各講師は、同じ時間に授業を 2 つ受け持つことはできない。
- カリキュラムの制約: 1 つのカリキュラムに、2 つの授業を同じ時間に設定することはできない。
- 部屋の占有: 同じ時間 (Period) の同じ講義室に、2 つの授業を割り当てることはできない。
- 利用不可の時間 (データセットごとに指定): 授業には割り当てられない時間がある。

ソフト制約:

- 講義室の収容人数: 講義室の収容人数は、その授業を受ける学生の数よりも多くなければならない。
- 最小限の就業日数: 同じコースの授業の開講期間は、最短になるようにする。
- カリキュラムの緊密さ: 同じカリキュラムに含まれる授業は、時間帯を近く (連続した時間に) 設定する。
- 講義室の不変性: 同じコースの授業は同じ講義室を割り当てる必要がある。

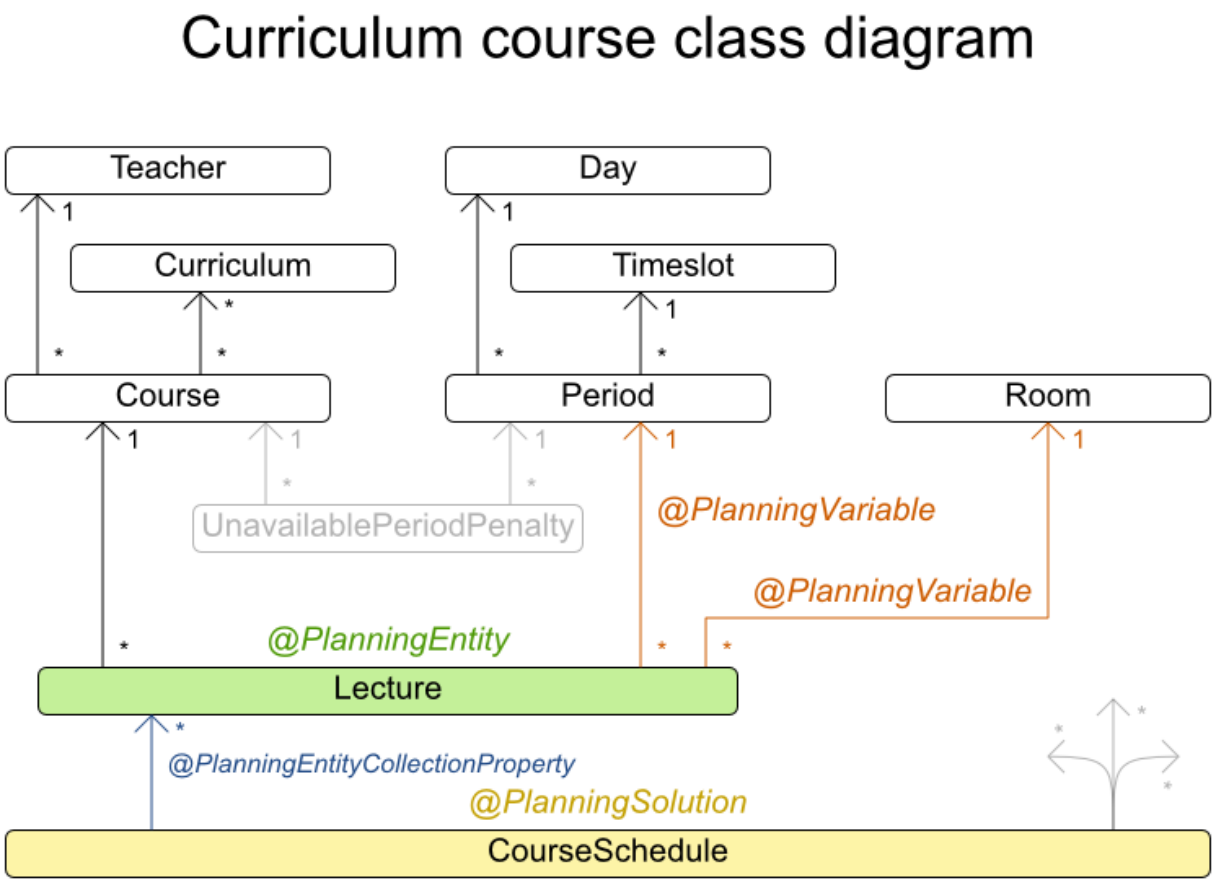
この問題は International Timetabling Competition 2007 track 3 で定義されています。

3.3.1.2. 問題の規模

> comp01 has 24 teachers, 14 curricula, 30 courses, 160 lectures, 30 periods, 6 rooms and 53 unavailable period constraints with a search space of 10^360.
> comp02 has 71 teachers, 70 curricula, 82 courses, 283 lectures, 25 periods, 16 rooms and 513 unavailable period constraints with a search space of 10^736.
> comp03 has 61 teachers, 68 curricula, 72 courses, 251 lectures, 25 periods, 16 rooms and 382 unavailable period constraints with a search space of 10^653.
> comp04 has 70 teachers, 57 curricula, 79 courses, 286 lectures, 25 periods, 18 rooms and 396 unavailable period constraints with a search space of 10^758.
> comp05 has 47 teachers, 139 curricula, 54 courses, 152 lectures, 36 periods, 9 rooms and 771 unavailable period constraints with a search space of 10^381.
> comp06 has 87 teachers, 70 curricula, 108 courses, 361 lectures, 25 periods, 18 rooms and 632 unavailable period constraints with a search space of 10^957.
> comp07 has 99 teachers, 77 curricula, 131 courses, 434 lectures, 25 periods, 20 rooms and 667 unavailable period constraints with a search space of 10^1171.
> comp08 has 76 teachers, 61 curricula, 86 courses, 324 lectures, 25 periods, 18 rooms and 478 unavailable period constraints with a search space of 10^859.
> comp09 has 68 teachers, 75 curricula, 76 courses, 279 lectures, 25 periods, 18 rooms and 405 unavailable period constraints with a search space of 10^740.
> comp10 has 88 teachers, 67 curricula, 115 courses, 370 lectures, 25 periods, 18 rooms and 694

```
unavailable period constraints with a search space of  10^981.
comp11 has 24 teachers,  13 curricula,  30 courses, 162 lectures, 45 periods,  5 rooms and   94
unavailable period constraints with a search space of  10^381.
comp12 has 74 teachers, 150 curricula,  88 courses, 218 lectures, 36 periods, 11 rooms and 1368
unavailable period constraints with a search space of  10^566.
comp13 has 77 teachers,  66 curricula,  82 courses, 308 lectures, 25 periods, 19 rooms and  468
unavailable period constraints with a search space of  10^824.
comp14 has 68 teachers,  60 curricula,  85 courses, 275 lectures, 25 periods, 17 rooms and  486
unavailable period constraints with a search space of  10^722.
```

3.3.1.3. ドメインモデル



3.3.2. マシンの再割当て (Google ROADEF 2012)

3.3.2.1. 問題の詳細

各プロセスをマシンに割り当てます。全プロセスには、すでに元の (最適化されていない) 割り当てがあります。プロセスにはそれぞれ、各リソース (CPU、メモリーなど) が一定量必要です。これは、クラウドのバランスの例の応用です。

ハード制約:

- 最大容量: マシンに割り当てる各リソースはこの量を超えてはいけない。
- 制約: 同じサービスのプロセスは別のマシンで実行する必要がある。
- 分散: 同じサービスのプロセスは複数の場所に分散させる必要がある。

- 依存関係: 他のサービスに依存するサービスのプロセスは、そのサービスの近くで実行する必要がある。
- 一時的な使用: リソースによっては一時的なものがあり、元のマシンと、新たに割り当てられたマシンの両方の最大容量にカウントされる。

ソフト制約:

- 負荷: 各マシンの各リソースの安全容量を超えてはいけない。
- 負荷分散: 各マシンで利用可能なリソースを分散させて、今後の割り当てに対応できるように容量を空ける。
- プロセスの移動コスト: プロセスには移動コストが発生する。
- サービスの移動コスト: サービスには移動コストが発生する。
- 機械の移動コスト: マシン A からマシン B にプロセスを移動すると、A から B に固有の移動コストが別途発生する。

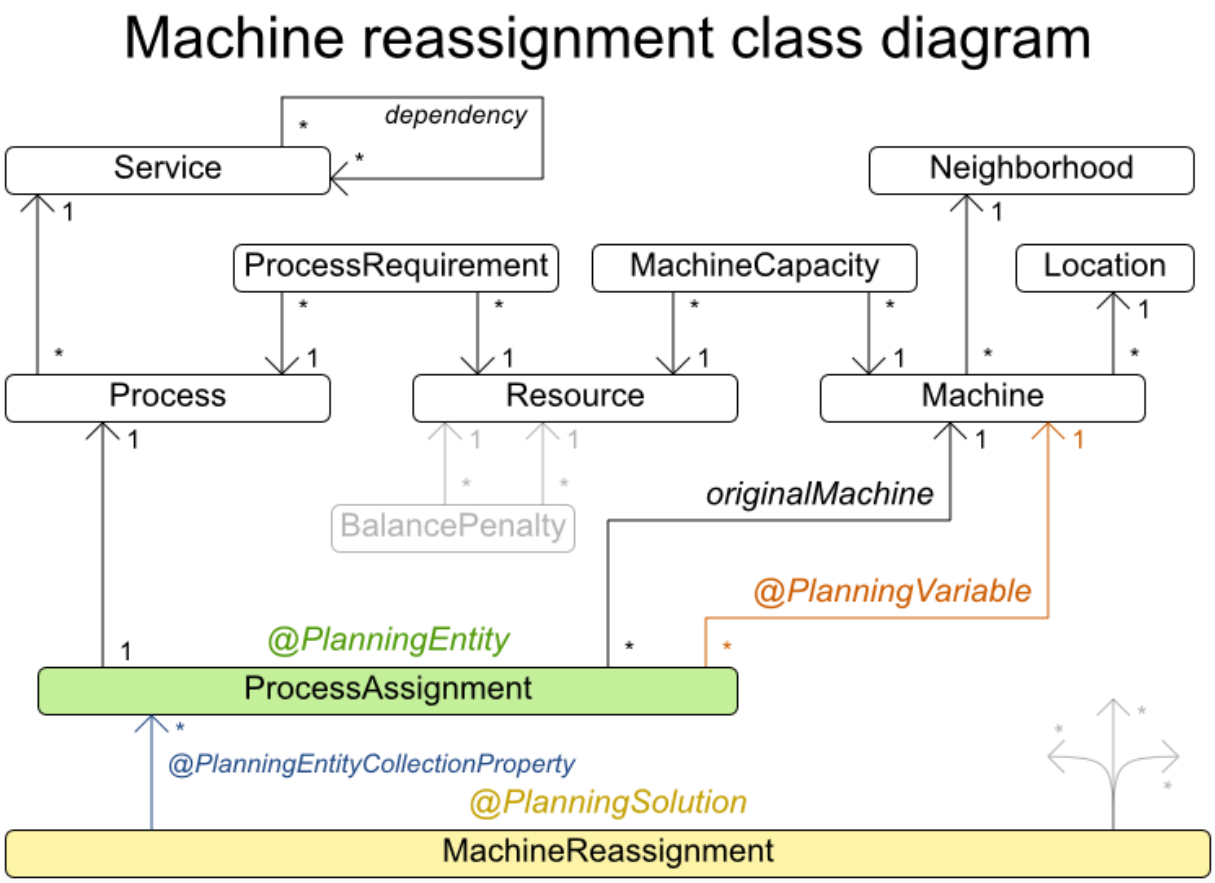
この問題は the Google ROADEF/EURO Challenge 2012 で定義されています。

3.3.2.2. 問題の規模

```
model_a1_1 has  2 resources,  1 neighborhoods,   4 locations,    4 machines,    79 services,   100
processes and 1 balancePenalties with a search space of     10^60.
model_a1_2 has  4 resources,  2 neighborhoods,   4 locations,  100 machines,   980 services,  1000
processes and 0 balancePenalties with a search space of   10^2000.
model_a1_3 has  3 resources,  5 neighborhoods,  25 locations,  100 machines,   216 services,  1000
processes and 0 balancePenalties with a search space of   10^2000.
model_a1_4 has  3 resources, 50 neighborhoods,  50 locations,   50 machines,   142 services,  1000
processes and 1 balancePenalties with a search space of   10^1698.
model_a1_5 has  4 resources,  2 neighborhoods,   4 locations,   12 machines,   981 services,  1000
processes and 1 balancePenalties with a search space of   10^1079.
model_a2_1 has  3 resources,  1 neighborhoods,   1 locations,  100 machines,  1000 services,  1000
processes and 0 balancePenalties with a search space of   10^2000.
model_a2_2 has 12 resources,  5 neighborhoods,  25 locations,  100 machines,   170 services,  1000
processes and 0 balancePenalties with a search space of   10^2000.
model_a2_3 has 12 resources,  5 neighborhoods,  25 locations,  100 machines,   129 services,  1000
processes and 0 balancePenalties with a search space of   10^2000.
model_a2_4 has 12 resources,  5 neighborhoods,  25 locations,   50 machines,   180 services,  1000
processes and 1 balancePenalties with a search space of   10^1698.
model_a2_5 has 12 resources,  5 neighborhoods,  25 locations,   50 machines,   153 services,  1000
processes and 0 balancePenalties with a search space of   10^1698.
model_b_1  has 12 resources,  5 neighborhoods,  10 locations,  100 machines,  2512 services,  5000
processes and 0 balancePenalties with a search space of  10^10000.
model_b_2  has 12 resources,  5 neighborhoods,  10 locations,  100 machines,  2462 services,  5000
processes and 1 balancePenalties with a search space of  10^10000.
model_b_3  has  6 resources,  5 neighborhoods,  10 locations,  100 machines, 15025 services, 20000
processes and 0 balancePenalties with a search space of  10^40000.
model_b_4  has  6 resources,  5 neighborhoods,  50 locations,  500 machines,  1732 services, 20000
processes and 1 balancePenalties with a search space of  10^53979.
model_b_5  has  6 resources,  5 neighborhoods,  10 locations,  100 machines, 35082 services, 40000
processes and 0 balancePenalties with a search space of  10^80000.
model_b_6  has  6 resources,  5 neighborhoods,  50 locations,  200 machines, 14680 services, 40000
processes and 1 balancePenalties with a search space of  10^92041.
```

> model_b_7 has 6 resources, 5 neighborhoods, 50 locations, 4000 machines, 15050 services, 40000 processes and 1 balancePenalties with a search space of 10^144082.
> model_b_8 has 3 resources, 5 neighborhoods, 10 locations, 100 machines, 45030 services, 50000 processes and 0 balancePenalties with a search space of 10^100000.
> model_b_9 has 3 resources, 5 neighborhoods, 100 locations, 1000 machines, 4609 services, 50000 processes and 1 balancePenalties with a search space of 10^150000.
> model_b_10 has 3 resources, 5 neighborhoods, 100 locations, 5000 machines, 4896 services, 50000 processes and 1 balancePenalties with a search space of 10^184948.

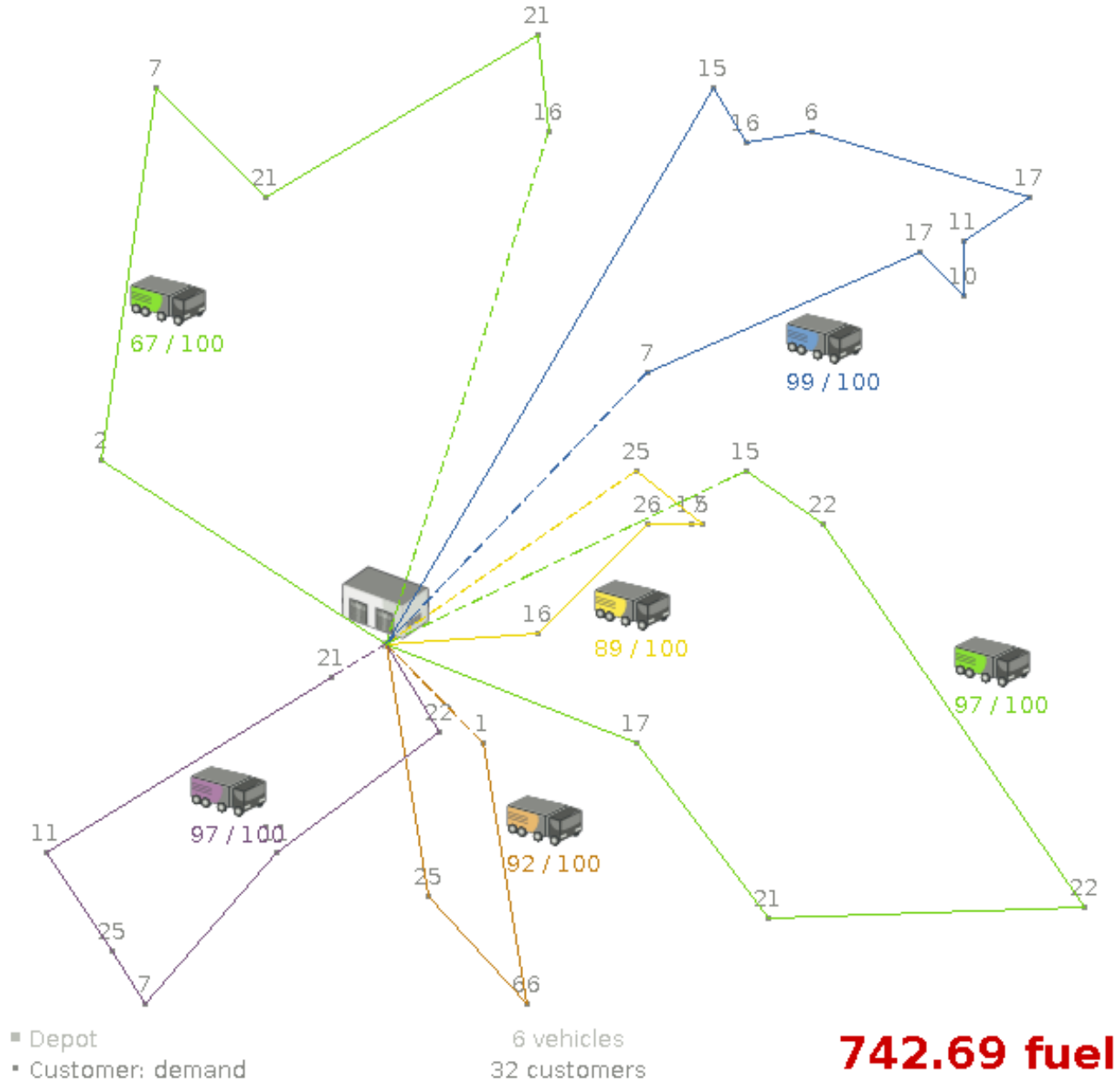
3.3.2.3. ドメインモデル



3.3.3. 配送経路

3.3.3.1. 問題の詳細

複数の車両を使用して、各顧客の品物を回収し、倉庫まで運びます。1つの車両で複数の顧客から品物を回収することはできますが、収容できる容量には限りがあります。



基本例 (CVRP) のほかに、時間枠の設定が加わった例 (CVRPTW) もあります。

ハード制約:

- 車両の容量: 車両は、車載容量を超えて品物を運ぶことができない。
- 時間枠 (CVRPTW のみ):
  - 移動時間: 別の場所に移動する場合には時間がかかる。
  - 顧客対応の時間: 車両は顧客に対応している時間、顧客先にとどまる必要がある。
  - 顧客の準備が整う時間: 顧客の準備が整う前に車両が到着する可能性があるが、準備ができるまで待機してから顧客に対応する必要がある。
  - 顧客が設定した締め切り時間: 車両は、顧客が設定した締め切り時間までに到着する必要がある。

ソフト制約:

- 合計距離: 車両が移動する合計距離 (ガソリンの消費量) を最小限に抑える。

CVRP (Capacitated Vehicle Routing Problem) と CVRPTW (Capacitated Vehicle Routing Problem with Time Window) は、VRP web で定義されています。

3.3.3.2. 問題の規模

CVRP インスタンス (時間枠なし):

```
A-n32-k5  has 1 depots,  5 vehicles and  31 customers with a search space of  10^46.
A-n33-k5  has 1 depots,  5 vehicles and  32 customers with a search space of  10^48.
A-n33-k6  has 1 depots,  6 vehicles and  32 customers with a search space of  10^48.
A-n34-k5  has 1 depots,  5 vehicles and  33 customers with a search space of  10^50.
A-n36-k5  has 1 depots,  5 vehicles and  35 customers with a search space of  10^54.
A-n37-k5  has 1 depots,  5 vehicles and  36 customers with a search space of  10^56.
A-n37-k6  has 1 depots,  6 vehicles and  36 customers with a search space of  10^56.
A-n38-k5  has 1 depots,  5 vehicles and  37 customers with a search space of  10^58.
A-n39-k5  has 1 depots,  5 vehicles and  38 customers with a search space of  10^60.
A-n39-k6  has 1 depots,  6 vehicles and  38 customers with a search space of  10^60.
A-n44-k7  has 1 depots,  7 vehicles and  43 customers with a search space of  10^70.
A-n45-k6  has 1 depots,  6 vehicles and  44 customers with a search space of  10^72.
A-n45-k7  has 1 depots,  7 vehicles and  44 customers with a search space of  10^72.
A-n46-k7  has 1 depots,  7 vehicles and  45 customers with a search space of  10^74.
A-n48-k7  has 1 depots,  7 vehicles and  47 customers with a search space of  10^78.
A-n53-k7  has 1 depots,  7 vehicles and  52 customers with a search space of  10^89.
A-n54-k7  has 1 depots,  7 vehicles and  53 customers with a search space of  10^91.
A-n55-k9  has 1 depots,  9 vehicles and  54 customers with a search space of  10^93.
A-n60-k9  has 1 depots,  9 vehicles and  59 customers with a search space of 10^104.
A-n61-k9  has 1 depots,  9 vehicles and  60 customers with a search space of 10^106.
A-n62-k8  has 1 depots,  8 vehicles and  61 customers with a search space of 10^108.
A-n63-k10 has 1 depots, 10 vehicles and  62 customers with a search space of 10^111.
A-n63-k9  has 1 depots,  9 vehicles and  62 customers with a search space of 10^111.
A-n64-k9  has 1 depots,  9 vehicles and  63 customers with a search space of 10^113.
A-n65-k9  has 1 depots,  9 vehicles and  64 customers with a search space of 10^115.
A-n69-k9  has 1 depots,  9 vehicles and  68 customers with a search space of 10^124.
A-n80-k10 has 1 depots, 10 vehicles and  79 customers with a search space of 10^149.
F-n135-k7 has 1 depots,  7 vehicles and 134 customers with a search space of 10^285.
F-n45-k4  has 1 depots,  4 vehicles and  44 customers with a search space of  10^72.
F-n72-k4  has 1 depots,  4 vehicles and  71 customers with a search space of 10^131.
```

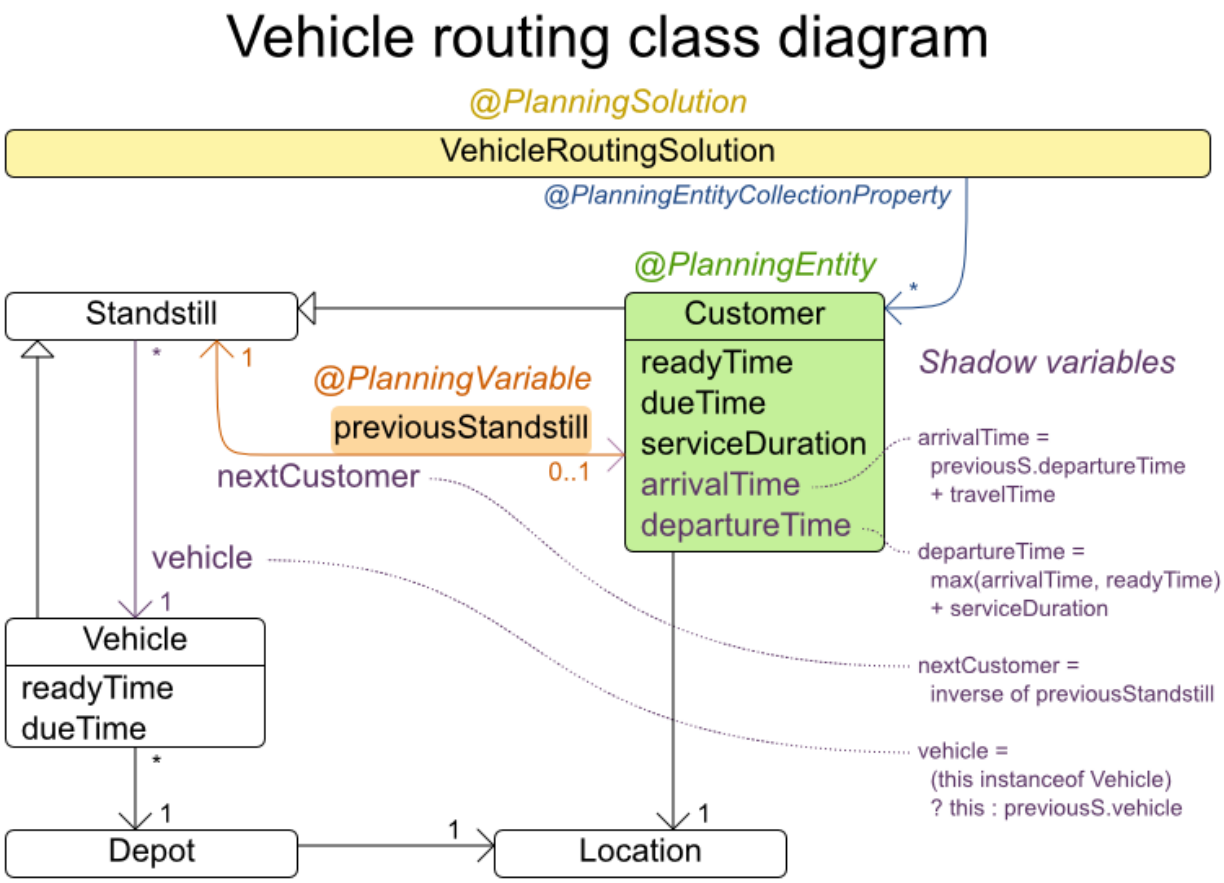
CVRPTW インスタンス (時間枠あり):

```
Solomon_025_C101       has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_025_C201       has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_025_R101       has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_025_R201       has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_025_RC101      has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_025_RC201      has 1 depots,  25 vehicles and   25 customers with a search space of
10^34.
Solomon_100_C101       has 1 depots,  25 vehicles and  100 customers with a search space of
10^200.
Solomon_100_C201       has 1 depots,  25 vehicles and  100 customers with a search space of
```

10^200.
Solomon_100_R101 has 1 depots, 25 vehicles and 100 customers with a search space of 10^200.
Solomon_100_R201 has 1 depots, 25 vehicles and 100 customers with a search space of 10^200.
Solomon_100_RC101 has 1 depots, 25 vehicles and 100 customers with a search space of 10^200.
Solomon_100_RC201 has 1 depots, 25 vehicles and 100 customers with a search space of 10^200.
Homberger_0200_C1_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0200_C2_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0200_R1_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0200_R2_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0200_RC1_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0200_RC2_2_1 has 1 depots, 50 vehicles and 200 customers with a search space of 10^460.
Homberger_0400_C1_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0400_C2_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0400_R1_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0400_R2_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0400_RC1_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0400_RC2_4_1 has 1 depots, 100 vehicles and 400 customers with a search space of 10^1040.
Homberger_0600_C1_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0600_C2_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0600_R1_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0600_R2_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0600_RC1_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0600_RC2_6_1 has 1 depots, 150 vehicles and 600 customers with a search space of 10^1666.
Homberger_0800_C1_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of 10^2322.
Homberger_0800_C2_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of 10^2322.
Homberger_0800_R1_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of 10^2322.
Homberger_0800_R2_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of 10^2322.
Homberger_0800_RC1_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of 10^2322.
Homberger_0800_RC2_8_1 has 1 depots, 200 vehicles and 800 customers with a search space of

> 10^2322.
> Homberger_1000_C110_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.
> Homberger_1000_C210_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.
> Homberger_1000_R110_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.
> Homberger_1000_R210_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.
> Homberger_1000_RC110_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.
> Homberger_1000_RC210_1 has 1 depots, 250 vehicles and 1000 customers with a search space of 10^3000.

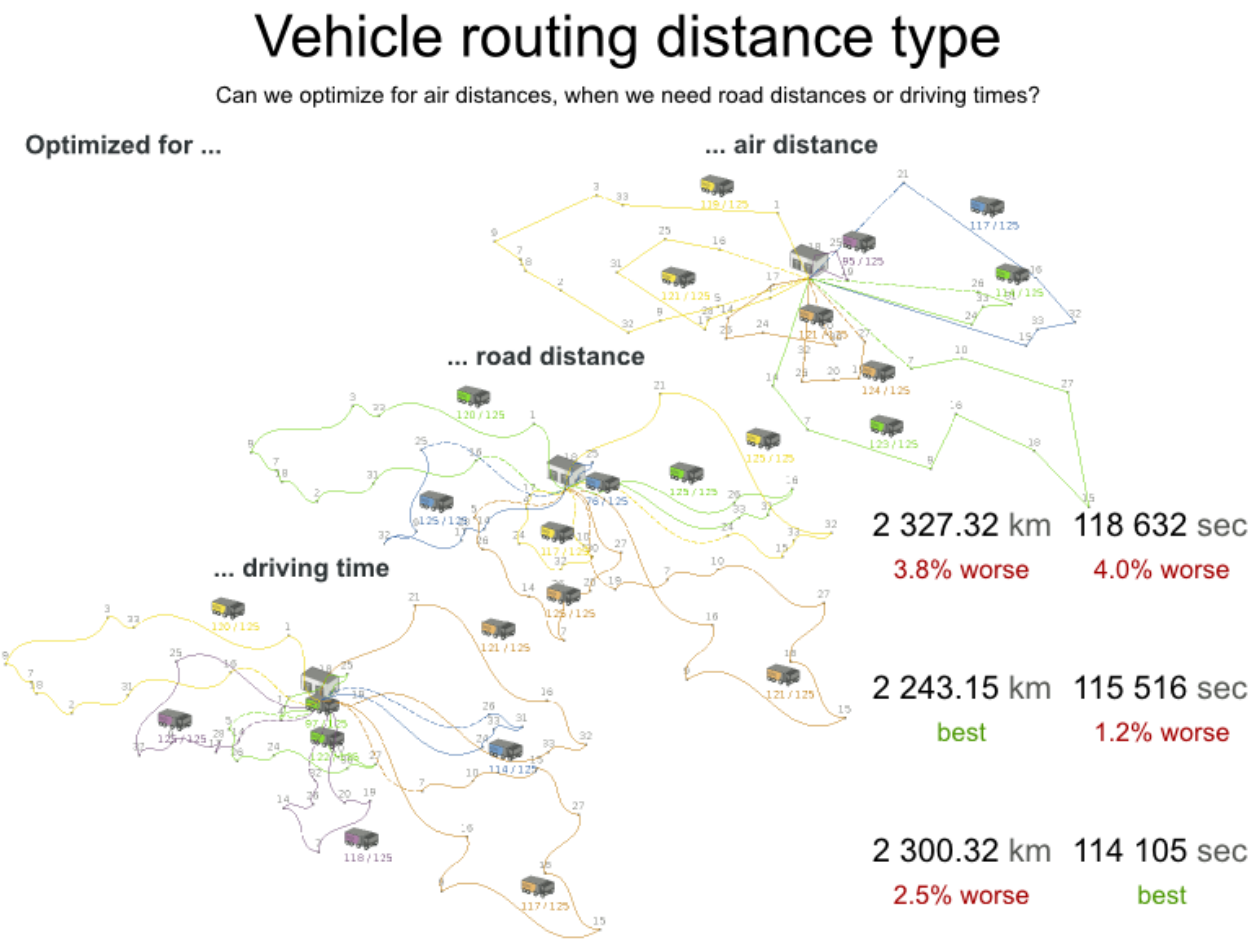
3.3.3.3. ドメインモデル



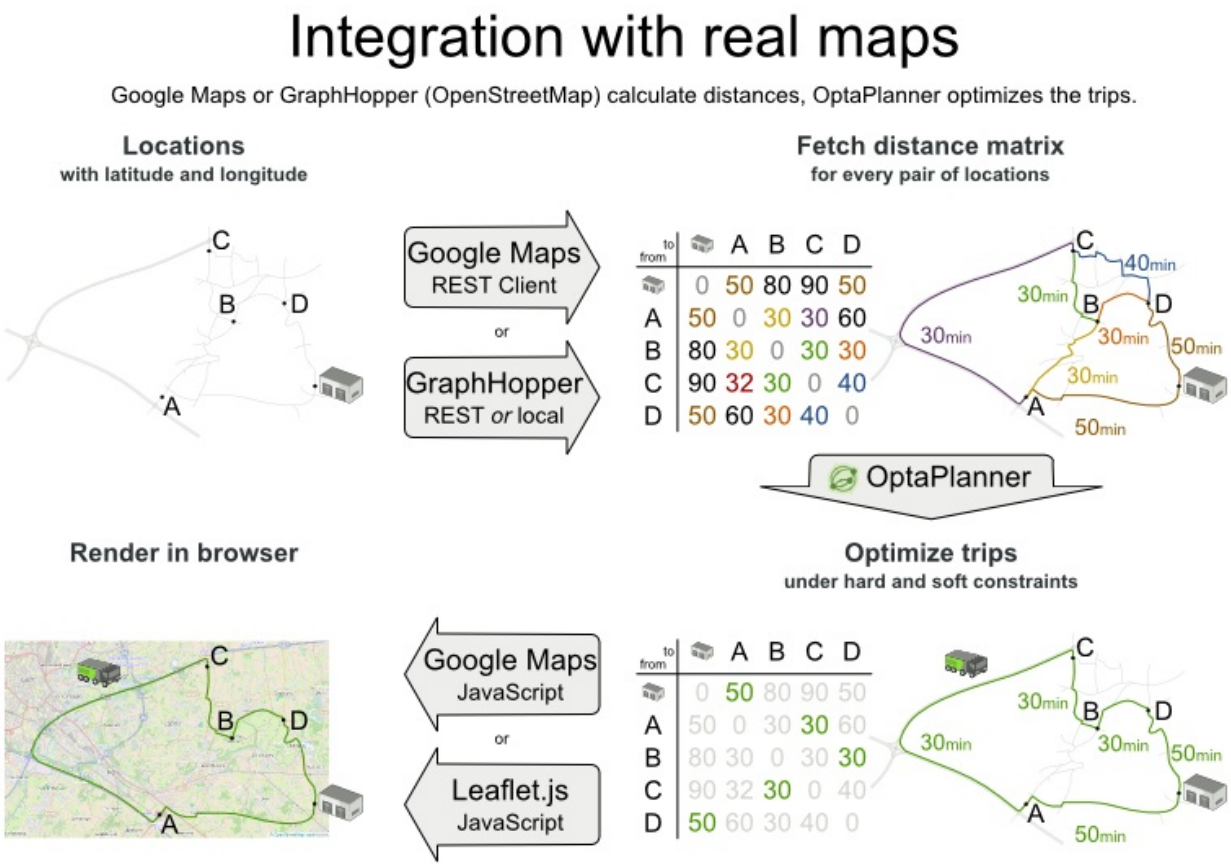
時間枠ありの配送経路のドメインモデルでは、シャドウ変数 を多用します。こうすることで、**arrivalTime** や **departureTime** などのプロパティーがドメインモデルで直接利用できるため、より自然に制約を表現することができます。

3.3.3.4. 直線距離ではなく道路の距離

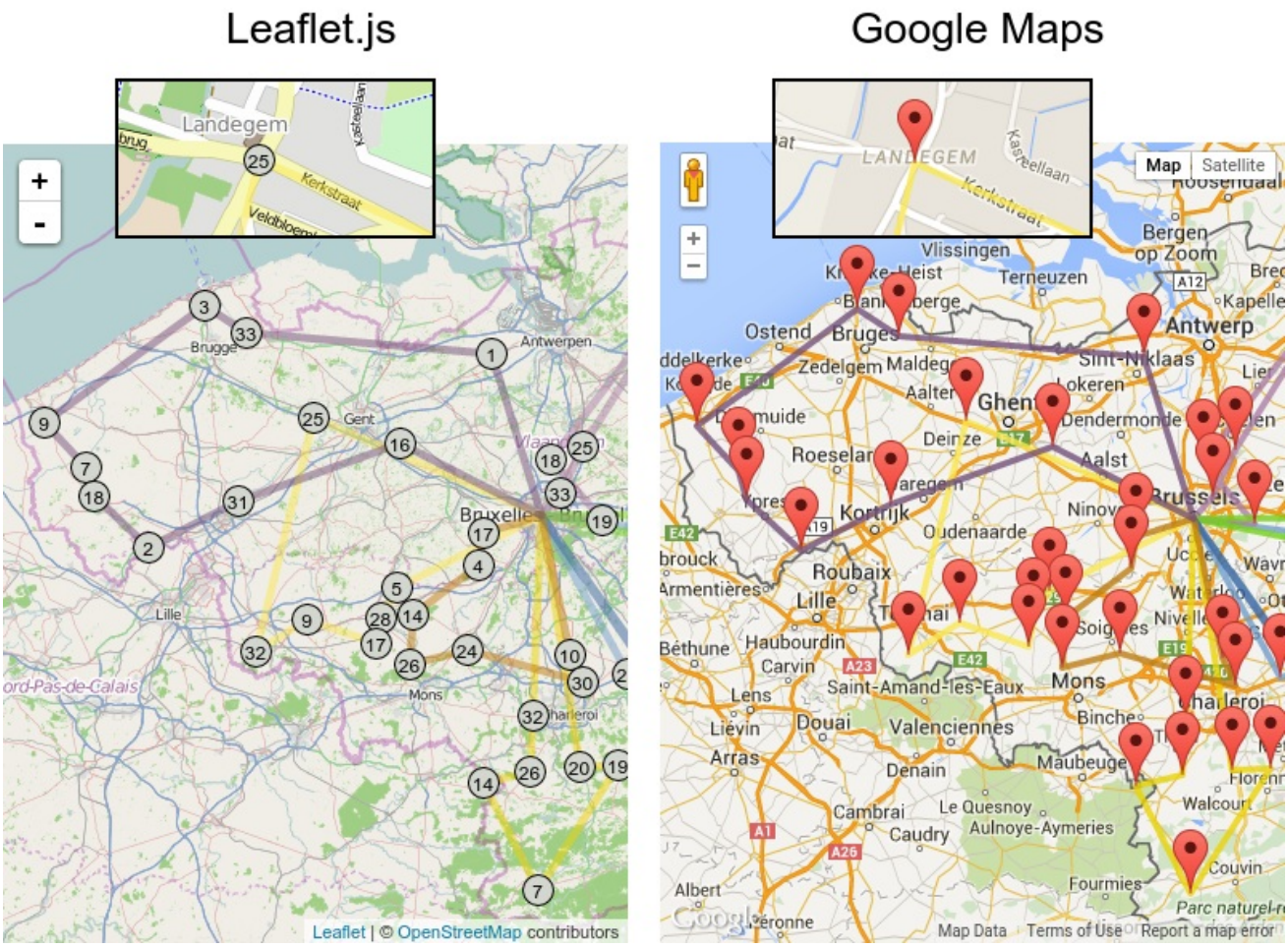
車は、直線距離を移動するのではなく、道路や高速道路を使用する必要があります。ビジネスの観点からすると、これは非常に重要です。



最適化アルゴリズムでは、2 点の距離を検索できている (できれば、事前に計算されている) 場合には、これは特に重要ではありません。移動のコストについては、距離の代わりに移動時間、ガソリン代、またはこれらの重み関数をベースにすることも可能です。GraphHopper (埋め込み可能なオフライン Java エンジン)、Open MapQuest (web サービス)、Google Maps Client API (web サービス) など、移動コストを事前に計算する技術があります。

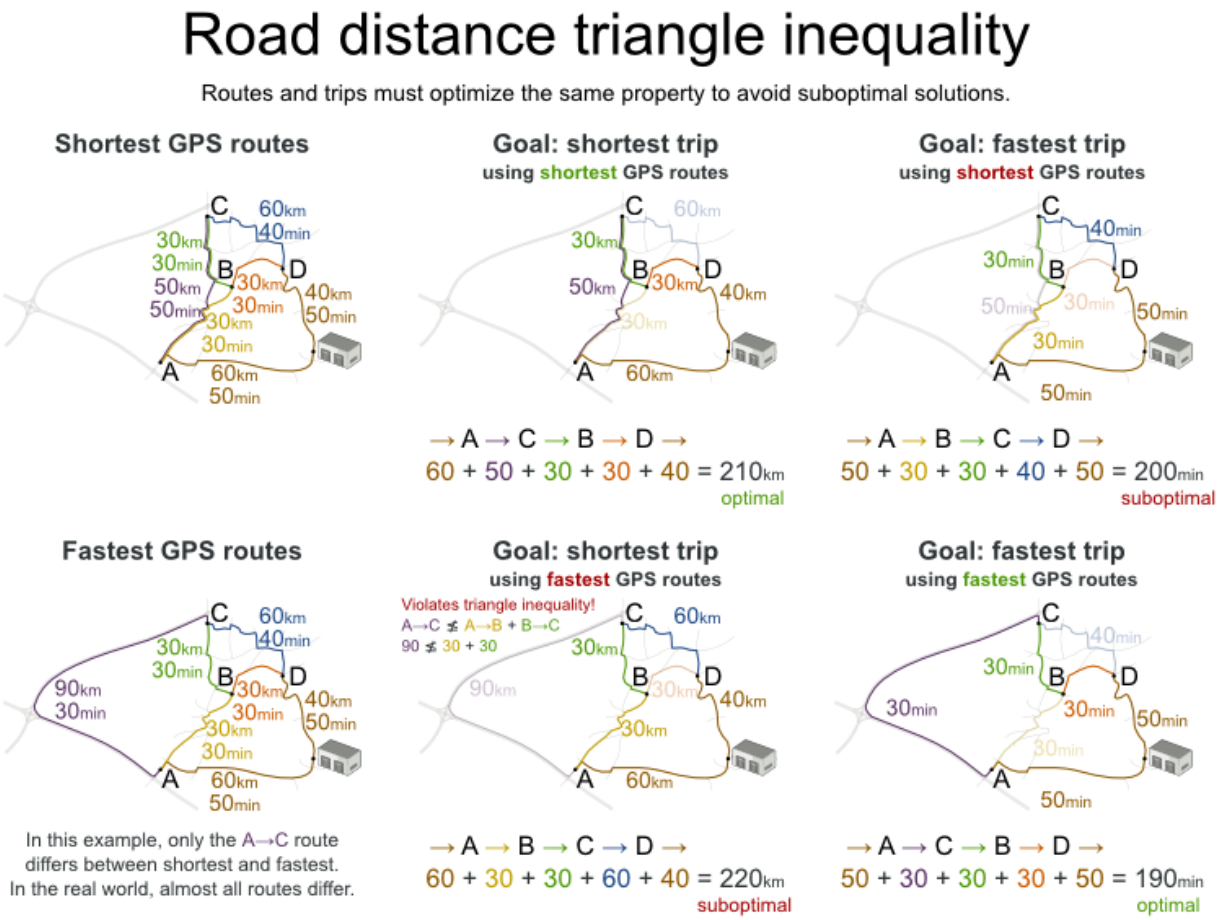


また、Leaflet や Google Maps for developers など、移動コストをレンダリングする技術もあります。**optaplanner-webexamples-*.war** には、このようなレンダリングのデモ例が含まれています。



GraphHopper または Google Map Directions を使用して実際の経路をレンダリングすることも可能ですが、高速道路で経路が重なるため、停止する順番を確認するのが困難になります。

2 点間の移動コストは、Planner で使用したのと同じ最適化条件を使用する点に注意してください。たとえば、GraphHopper はデフォルトで、最短ではなく、最速の経路を返します。「最速」の GPS 経路の km (またはマイル) の距離を使用して、Planner で「最短」の移動を最適化しないようにしてください。以下のように、準最適な解が導き出される可能性があります。

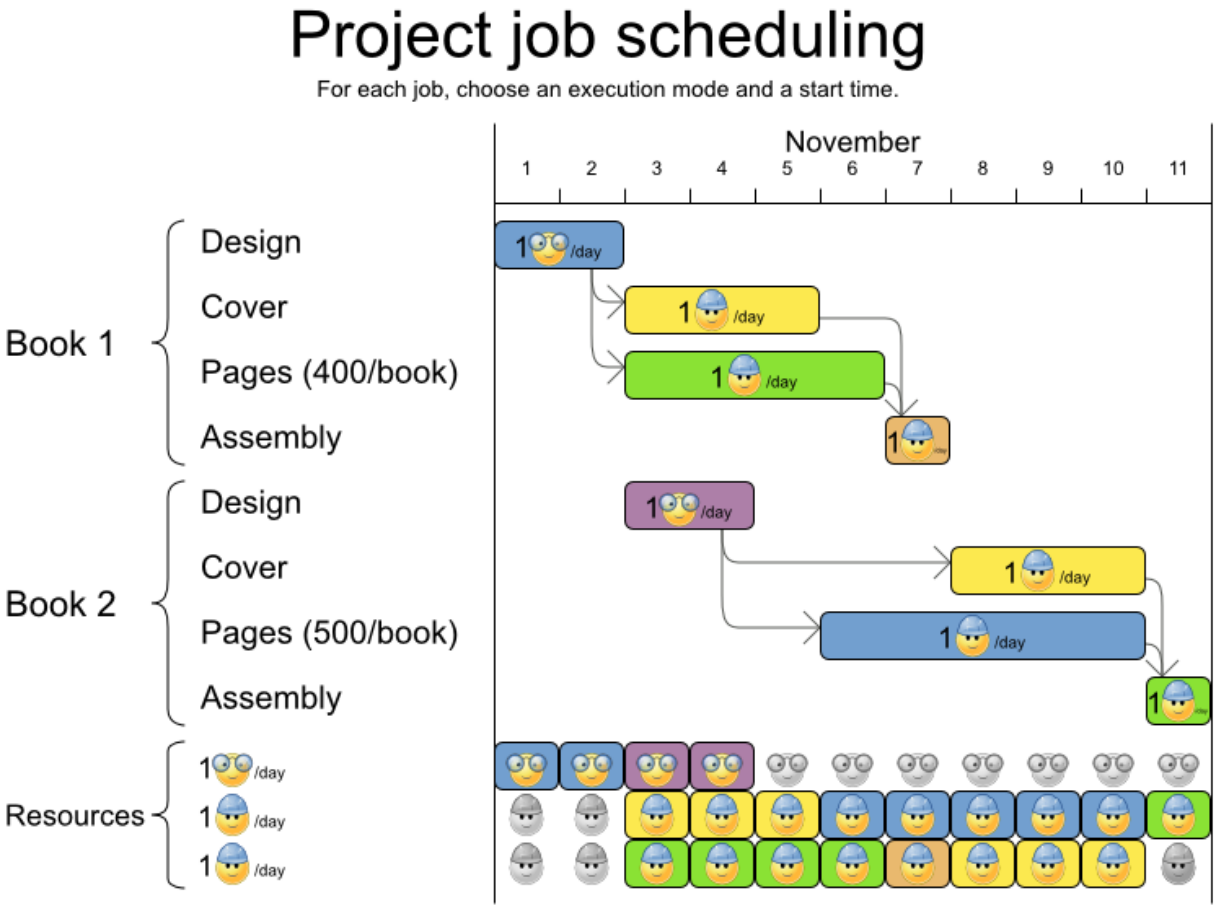


一般的な考え方とは異なり、多くのユーザーは最短の経路ではなく、最速の経路を使用したいと考えます。通常の道路よりも高速道路の使用を、舗装されていない道よりも舗装されている道路を好みます。実際には、最速の経路と、最短の経路が同じであることはほぼありません。

3.3.4. プロジェクトジョブのスケジュール

3.3.4.1. 問題の詳細

プロジェクトの遅延を最小限に抑えるために、すべてのジョブを時間内に実行できるようにスケジュールを設定します。各ジョブは、プロジェクトに含まれます。ジョブは、異なる方法で実行でき、方法ごとに期間や使用するリソースが異なります。これは、柔軟な ジョブショップスケジューリング (JSP) の応用です。



ハード制約:

- ジョブの優先順位: ジョブは、先行のジョブがすべて完了するまで開始しない。
- リソースの容量: 利用可能な量を超えるリソースを使用しない。
  - リソースはローカル (同じプロジェクトのジョブ間で共有)、またはグローバル (全ジョブ間で共有) とする。
  - リソースは更新可能 (1 日に利用可能な容量) または更新不可 (全日で利用可能な容量) とする。

中程度の制約:

- プロジェクトの合計遅延時間: 各プロジェクトの所要時間 (メイクスパン) を最短にする。

ソフト制約:

- メイクスパン合計: 複数のプロジェクトスケジュールの合計所要時間を最短にする。

この問題は MISTA 2013 challenge で定義されています。

3.3.4.2. 問題の規模

> Schedule A-1 has 2 projects, 24 jobs, 64 execution modes, 7 resources and 150 resource requirements.
> Schedule A-2 has 2 projects, 44 jobs, 124 execution modes, 7 resources and 420 resource requirements.

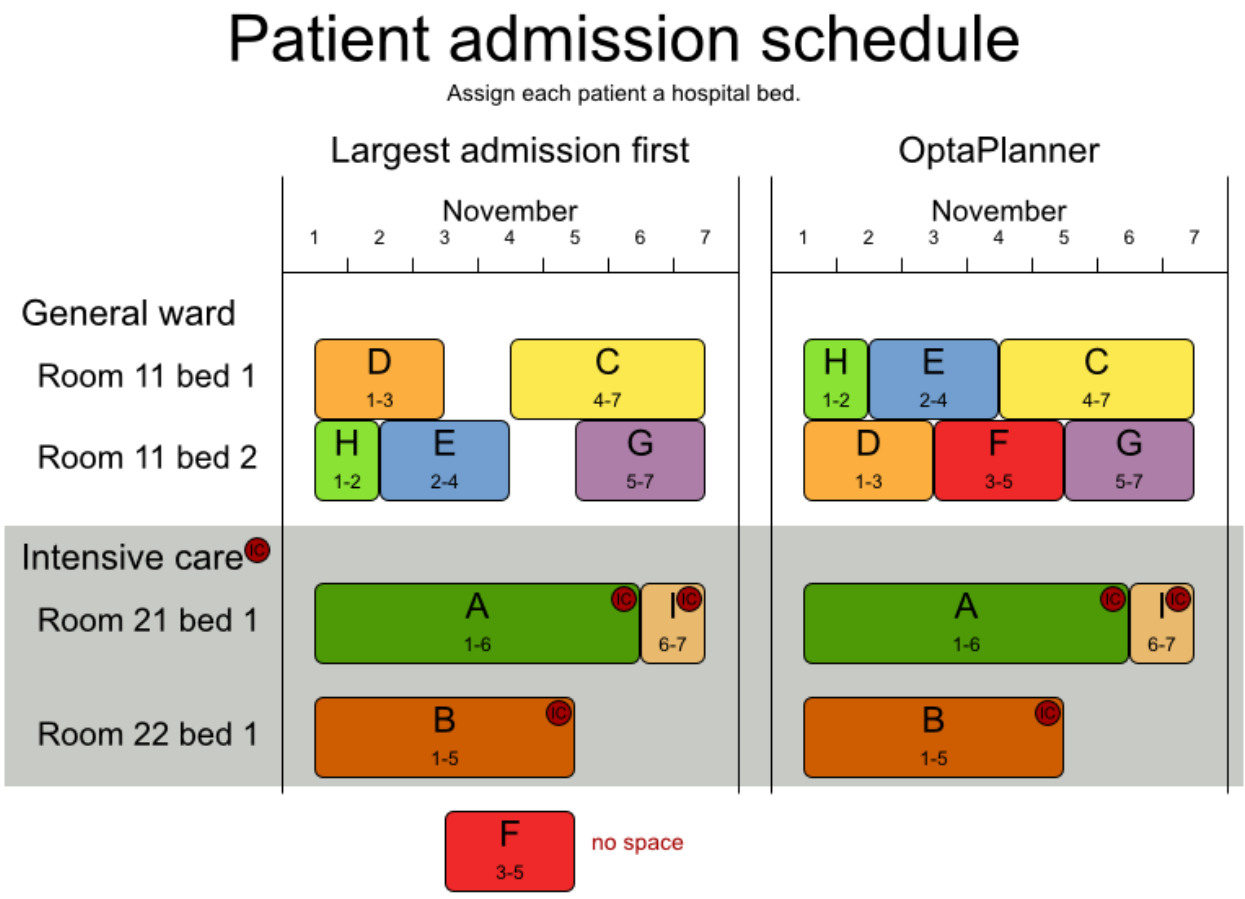
Schedule A-3 has 2 projects, 64 jobs, 184 execution modes, 7 resources and 630 resource requirements.
Schedule A-4 has 5 projects, 60 jobs, 160 execution modes, 16 resources and 390 resource requirements.
Schedule A-5 has 5 projects, 110 jobs, 310 execution modes, 16 resources and 900 resource requirements.
Schedule A-6 has 5 projects, 160 jobs, 460 execution modes, 16 resources and 1440 resource requirements.
Schedule A-7 has 10 projects, 120 jobs, 320 execution modes, 22 resources and 900 resource requirements.
Schedule A-8 has 10 projects, 220 jobs, 620 execution modes, 22 resources and 1860 resource requirements.
Schedule A-9 has 10 projects, 320 jobs, 920 execution modes, 31 resources and 2880 resource requirements.
Schedule A-10 has 10 projects, 320 jobs, 920 execution modes, 31 resources and 2970 resource requirements.
Schedule B-1 has 10 projects, 120 jobs, 320 execution modes, 31 resources and 900 resource requirements.
Schedule B-2 has 10 projects, 220 jobs, 620 execution modes, 22 resources and 1740 resource requirements.
Schedule B-3 has 10 projects, 320 jobs, 920 execution modes, 31 resources and 3060 resource requirements.
Schedule B-4 has 15 projects, 180 jobs, 480 execution modes, 46 resources and 1530 resource requirements.
Schedule B-5 has 15 projects, 330 jobs, 930 execution modes, 46 resources and 2760 resource requirements.
Schedule B-6 has 15 projects, 480 jobs, 1380 execution modes, 46 resources and 4500 resource requirements.
Schedule B-7 has 20 projects, 240 jobs, 640 execution modes, 61 resources and 1710 resource requirements.
Schedule B-8 has 20 projects, 440 jobs, 1240 execution modes, 42 resources and 3180 resource requirements.
Schedule B-9 has 20 projects, 640 jobs, 1840 execution modes, 61 resources and 5940 resource requirements.
Schedule B-10 has 20 projects, 460 jobs, 1300 execution modes, 42 resources and 4260 resource requirements.

3.3.5. 病床計画 (PAS - 入院スケジュール)

3.3.5.1. 問題の詳細

(入院予定の) 各患者に、入院時のベッドを割り当てます。各ベッドは病室に所属し、各病室は診療科部門に所属します。患者の入院日と退院日は固定されており、入院日のベッド割り当てだけを行う必要があります。

この問題は 過制約 データセットを使用しています。



ハード制約:

- 患者 2 名を、同じ日に、同じベッドに割り当てることはできない。重み **-1000hard * conflictNightCount**
- 病室には、性別の制約を加えることができる:1晩に割り当てる性別を女性のみ、または男性のみ (もしくは、性別の制約なし) に設定できる。重み: **-50hard * nightCount**
- 部門には、年齢の上限または下限を設定できる。重み: **-100hard * nightCount**
- 患者は、特定の設備のある病室をリクエストできる。重み: **-50hard * nightCount**

中程度の制約:

- データセットに制約を過剰に課さない場合は、すべての患者にベッドを割り当てる。重み: **-1medium * nightCount**

ソフト制約:

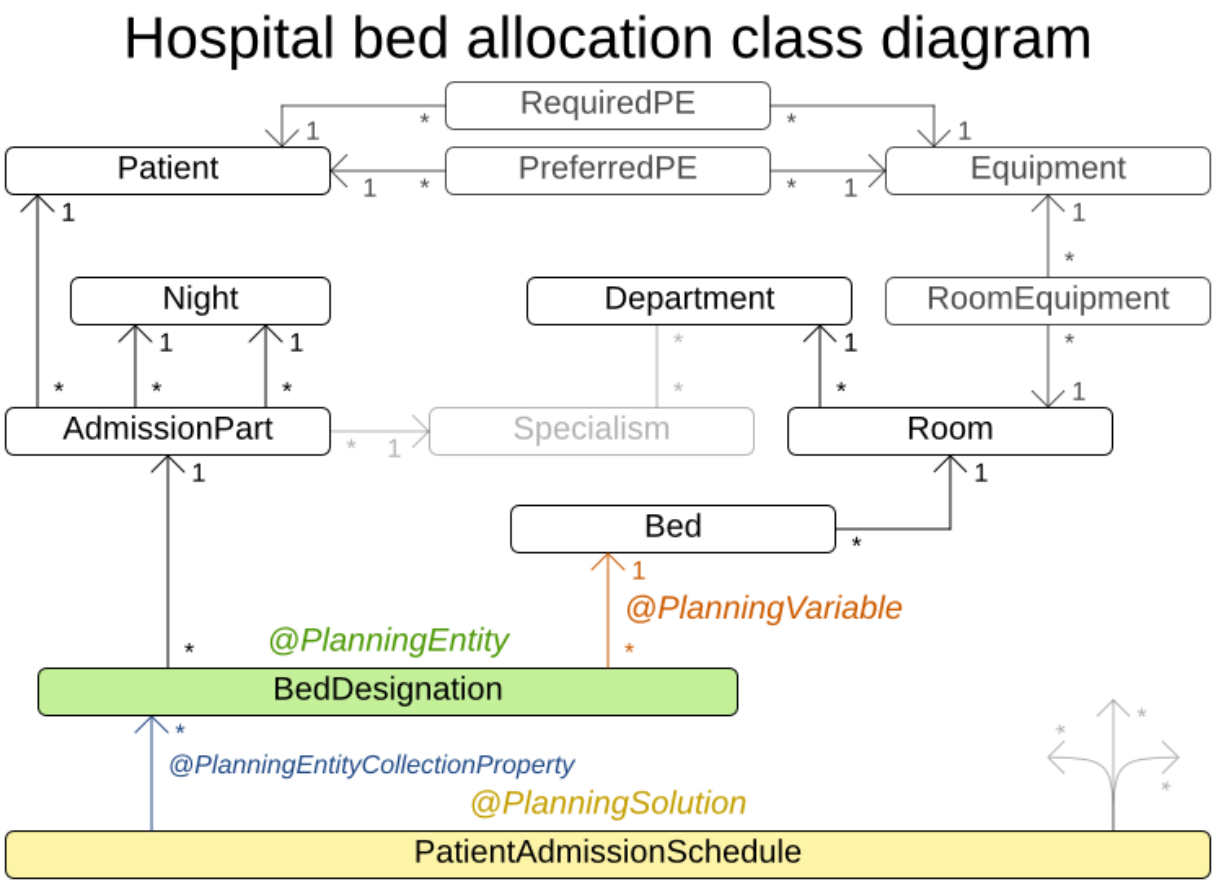
- 患者は、最大の病室サイズ (例: 一人部屋など) を選択できる。重み: **-8soft * nightCount**
- 患者は、その病状を専門とする診療科部門にできるだけ割り当てる。重み: **-10soft * nightCount**
- 患者は、その病状を専門とする病室にできるだけ割り当てる。重み: **-20soft * nightCount**
  - 病室の専門科の優先順位を 1 とする。重み: **-10soft * (priority - 1) * nightCount**
- 患者は特定の設備のある病室を希望することができる。重み: **-50hard * nightCount**

この問題は、Kaho Patient Scheduling に変更を加えたもので、データセットは病院から提供された実際のデータを使用しています。

3.3.5.2. 問題の規模

> testdata01 has 4 specialisms, 2 equipments, 4 departments, 98 rooms, 286 beds, 14 nights, 652 patients and 652 admissions with a search space of 10^1601.
> testdata02 has 6 specialisms, 2 equipments, 6 departments, 151 rooms, 465 beds, 14 nights, 755 patients and 755 admissions with a search space of 10^2013.
> testdata03 has 5 specialisms, 2 equipments, 5 departments, 131 rooms, 395 beds, 14 nights, 708 patients and 708 admissions with a search space of 10^1838.
> testdata04 has 6 specialisms, 2 equipments, 6 departments, 155 rooms, 471 beds, 14 nights, 746 patients and 746 admissions with a search space of 10^1994.
> testdata05 has 4 specialisms, 2 equipments, 4 departments, 102 rooms, 325 beds, 14 nights, 587 patients and 587 admissions with a search space of 10^1474.
> testdata06 has 4 specialisms, 2 equipments, 4 departments, 104 rooms, 313 beds, 14 nights, 685 patients and 685 admissions with a search space of 10^1709.
> testdata07 has 6 specialisms, 4 equipments, 6 departments, 162 rooms, 472 beds, 14 nights, 519 patients and 519 admissions with a search space of 10^1387.
> testdata08 has 6 specialisms, 4 equipments, 6 departments, 148 rooms, 441 beds, 21 nights, 895 patients and 895 admissions with a search space of 10^2366.
> testdata09 has 4 specialisms, 4 equipments, 4 departments, 105 rooms, 310 beds, 28 nights, 1400 patients and 1400 admissions with a search space of 10^3487.
> testdata10 has 4 specialisms, 4 equipments, 4 departments, 104 rooms, 308 beds, 56 nights, 1575 patients and 1575 admissions with a search space of 10^3919.
> testdata11 has 4 specialisms, 4 equipments, 4 departments, 107 rooms, 318 beds, 91 nights, 2514 patients and 2514 admissions with a search space of 10^6291.
> testdata12 has 4 specialisms, 4 equipments, 4 departments, 105 rooms, 310 beds, 84 nights, 2750 patients and 2750 admissions with a search space of 10^6851.
> testdata13 has 5 specialisms, 4 equipments, 5 departments, 125 rooms, 368 beds, 28 nights, 907 patients and 1109 admissions with a search space of 10^2845.

3.3.5.3. ドメインモデル

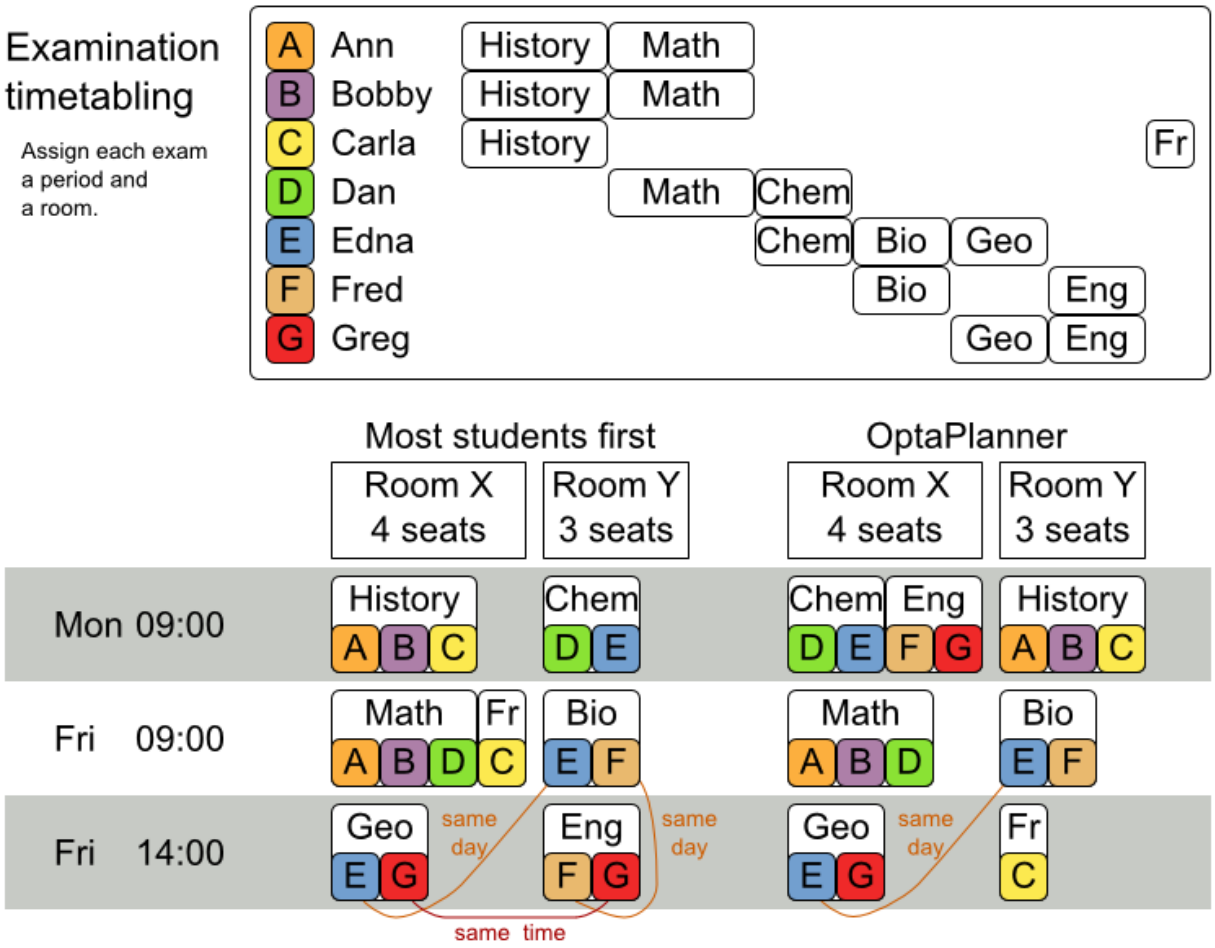


3.4. 複雑な例

3.4.1. 試験の時間割 (ITC 2007 track 1 - 試験)

3.4.1.1. 問題の詳細

すべての試験に、時間と部屋を割り当てます。同じ時間帯の同じ部屋で、複数の試験を行うことができるものとします。



ハード制約:

- 試験の制約: 同じ学生が受ける 2 つの試験は、同じ時間帯に実施できないものとする。
- 教室の収容人数: 教室の座席数は、常に受験者数よりも多くなければならない。
- 期間: 期間は、すべての試験に対応できる長さでなければならない。
- 期間関連のハード制約 (データセットごとに指定):
  - 一致: 特定の 2 つの試験を同じ時間帯に設定する必要がある (別の教室を使用することも可能)。
  - 除外: 特定の 2 つの試験を同じ時間帯に設定できない。
  - 以降: 特定の試験を、特定の試験の後に行う必要がある。
- 教室関連の制約 (データセット毎ごとに指定):
  - 排他的: 特定の試験を、他の試験と同じ教室で行うことはできない。

ソフト制約 (パラメーター化されたペナルティーがそれぞれ設定されている):

- 同じ学生が、続けて試験を 2 つ受けてはいけない。
- 同じ学生が、同じ日に試験を 2 つ受けてはいけない。
- 時間帯の分散: 同じ学生が受ける 2 つの試験は、時間をある程度あける。

- 異なる試験の長さ: 教室を共有する 2 つの試験の長さは、同じにする。
- 前倒し: 規模の大きい試験は、スケジュールを早めに決定する。
- 期間のペナルティー (データセットごとに指定): 期間によっては、使用されるとペナルティーが発生する。
- 部屋のペナルティー (データセットごとに指定): 部屋によっては、使用されるとペナルティーが発生する。

大学から取得した試験の大規模データセットを使用します。

この問題は International Timetabling Competition 2007 track 1 で明示されました。Geoffrey De Smet 氏は、ごく初期段階の Planner を使用して、このコンペティションで 4 位を獲得しました。このコンペティション以降、多くの改良点が加えられています。

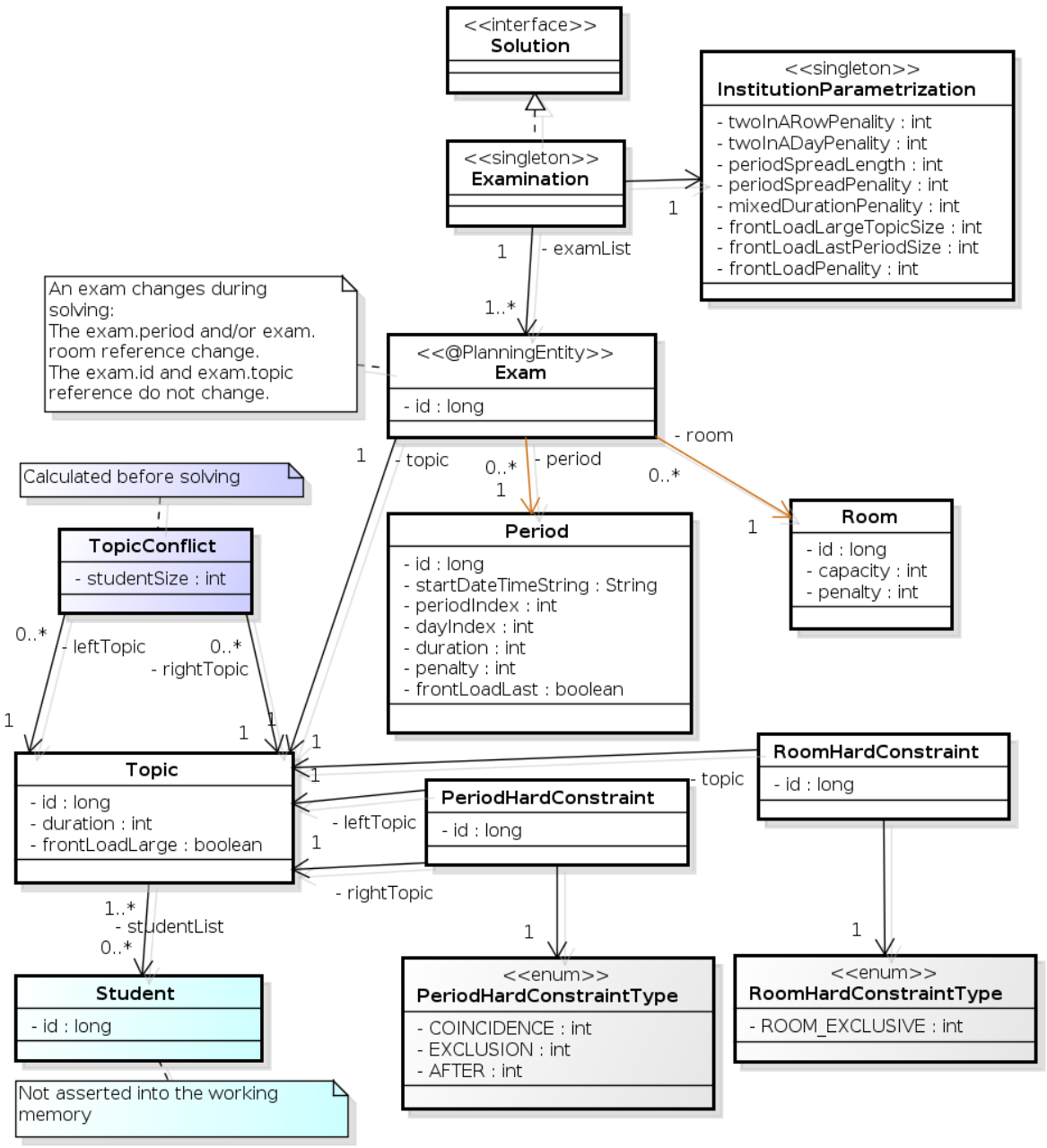
3.4.1.2. 問題の規模

> exam_comp_set1 has 7883 students, 607 exams, 54 periods, 7 rooms, 12 period constraints and 0 room constraints with a search space of 10^1564.
> exam_comp_set2 has 12484 students, 870 exams, 40 periods, 49 rooms, 12 period constraints and 2 room constraints with a search space of 10^2864.
> exam_comp_set3 has 16365 students, 934 exams, 36 periods, 48 rooms, 168 period constraints and 15 room constraints with a search space of 10^3023.
> exam_comp_set4 has 4421 students, 273 exams, 21 periods, 1 rooms, 40 period constraints and 0 room constraints with a search space of 10^360.
> exam_comp_set5 has 8719 students, 1018 exams, 42 periods, 3 rooms, 27 period constraints and 0 room constraints with a search space of 10^2138.
> exam_comp_set6 has 7909 students, 242 exams, 16 periods, 8 rooms, 22 period constraints and 0 room constraints with a search space of 10^509.
> exam_comp_set7 has 13795 students, 1096 exams, 80 periods, 15 rooms, 28 period constraints and 0 room constraints with a search space of 10^3374.
> exam_comp_set8 has 7718 students, 598 exams, 80 periods, 8 rooms, 20 period constraints and 1 room constraints with a search space of 10^1678.

3.4.1.3. ドメインモデル

以下に、主な試験のドメインクラスを紹介しています。

図3.3 試験のドメインクラスの図

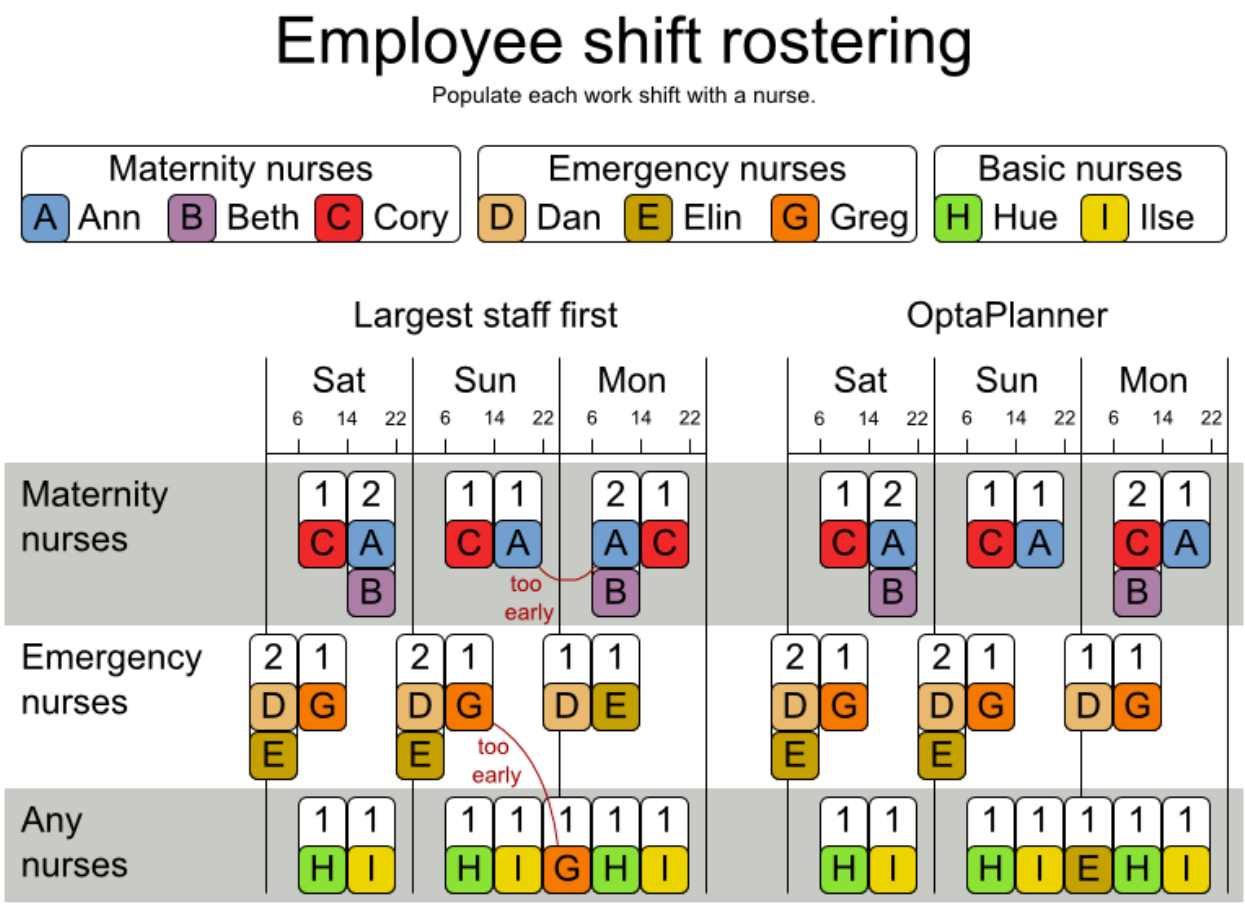


試験のコンセプトを、**Exam** クラスと **Topic** クラスに分けた点に注意してください。期間または教室のプロパティーを変更し、ソリューション (プランニングエンティティークラス) を求めると、**Exam** インスタンスが変化します。このとき、**Topic** インスタンス、**Period** インスタンス、および **Room** インスタンスは変化しません (他のクラスと同様、これらも問題ファクトです)。

3.4.2. 従業員の勤務表 (INRC 2010 - 看護師の勤務表)

3.4.2.1. 問題の詳細

各シフトに看護師を割り当てます。



ハード制約:

- 未割り当てのシフトなし (組み込み): すべてのシフトを従業員に割り当てる必要がある。
- シフトの制約: 従業員には 1 日に 1 シフトだけ割り当てることができる。

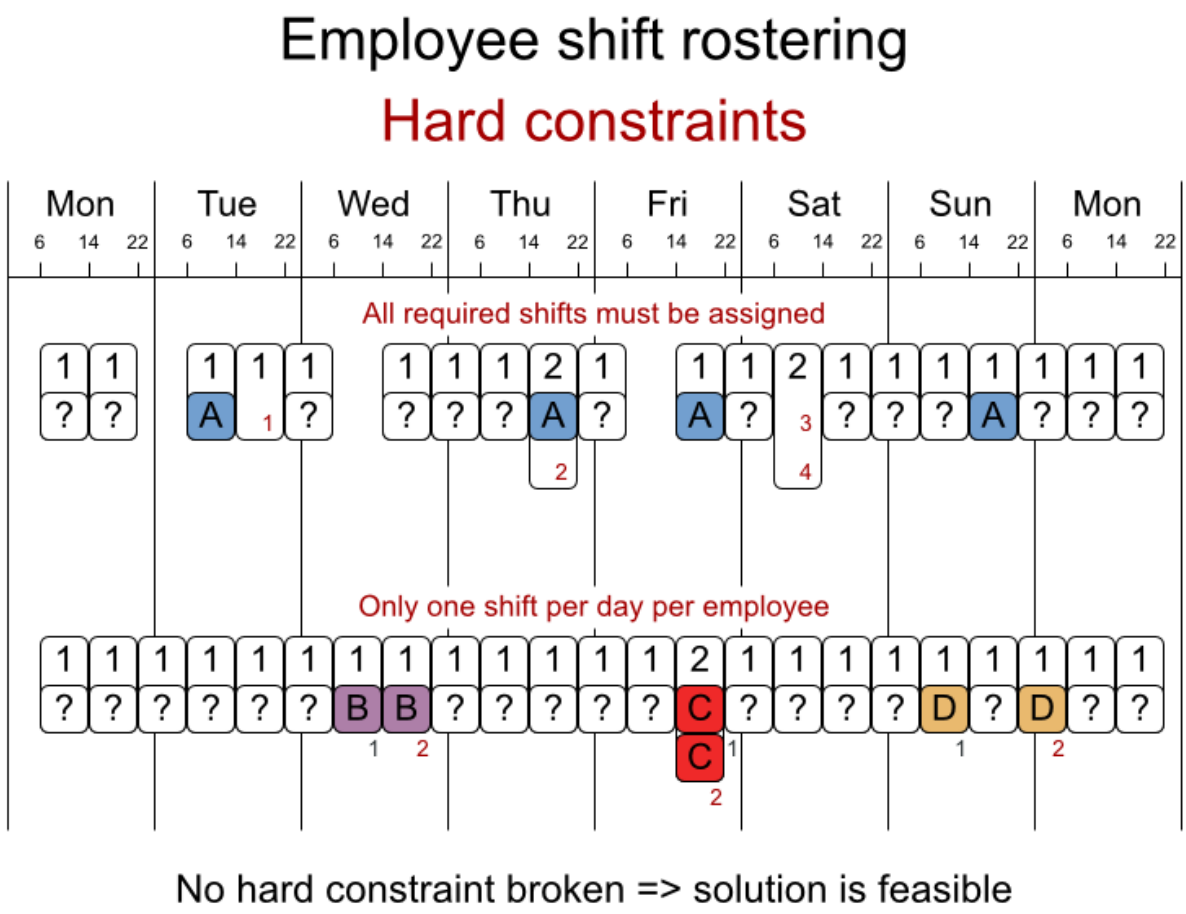
ソフト制約:

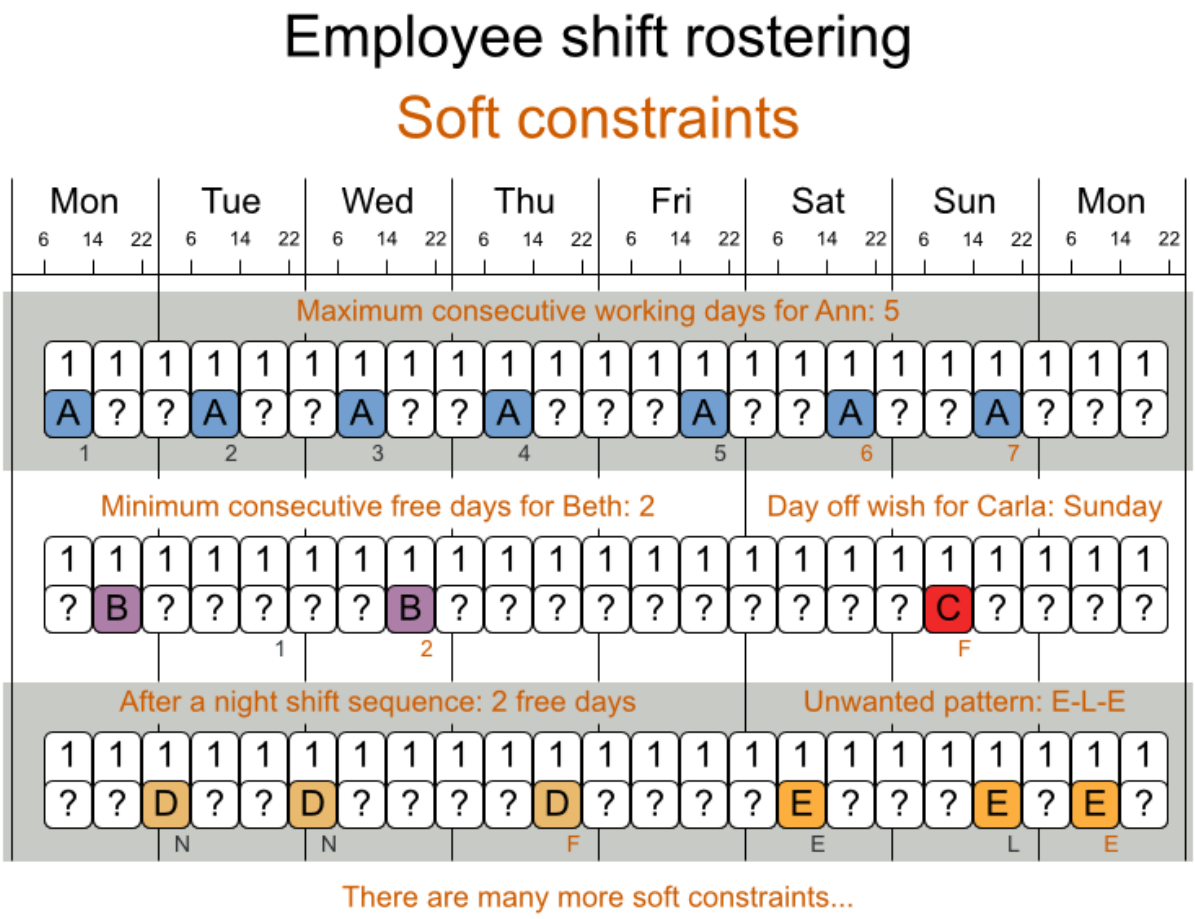
- 契約上の義務: この業界では、頻繁に契約上の義務に違反するため、ハード制約ではなく、ソフト制約として定義することに決定しました。
  - 割り当ての下限および上限: 各従業員は、(それぞれの契約に合わせて) x より多く、y よりも少ないシフト数を勤務する必要がある。
  - 連続勤務日数の下限および上限: 各従業員は、(それぞれの契約に合わせて) 連続で x 日から y 日間、勤務する必要がある。
  - 連続公休日数の下限および上限: 各従業員は、(それぞれの契約に合わせて) 連続で x 日から y 日間、休む必要がある。
  - 週末に連続勤務する回数の下限および上限: 各従業員は、(それぞれの契約に合わせて) 連続で x 回から y 回、週末勤務する必要がある。
  - 週末の勤務有無を同じにする: 各従業員は、週末の両日を勤務する、または休む必要がある。
  - 週末のシフトタイプを同じにする: 各従業員に、同じ週末のシフトタイプは、同じにする必要がある。

- 好ましくないシフトパターン: 遅番＋早番＋遅番など、好ましくないシフトタイプを連続で組み合わせたパターン。

- 従業員の希望:
  - 勤務日のリクエスト: 従業員は、特定の勤務希望日を申請できる。
  - 公休日のリクエスト: 従業員は、特定の公休希望日を申請できる。
  - 勤務するシフトのリクエスト: 従業員は特定のシフトへの割り当てを希望できる。
  - 勤務しないシフトのリクエスト: 従業員は特定のシフトに割り当てられないように希望できる。

- 他のスキル: シフトに割り当てられた従業員は、そのシフトで必要な全スキルに堪能である必要がある。





この問題は International Nurse Rostering Competition 2010 で定義されています。

3.4.2.2. 問題の規模

以下のように、データセットの種類は 3 つあります。

- sprint: 数秒で問題を解決する必要があります。
- medium: 数分で問題を解決する必要があります。
- long: 数時間で問題を解決する必要があります。

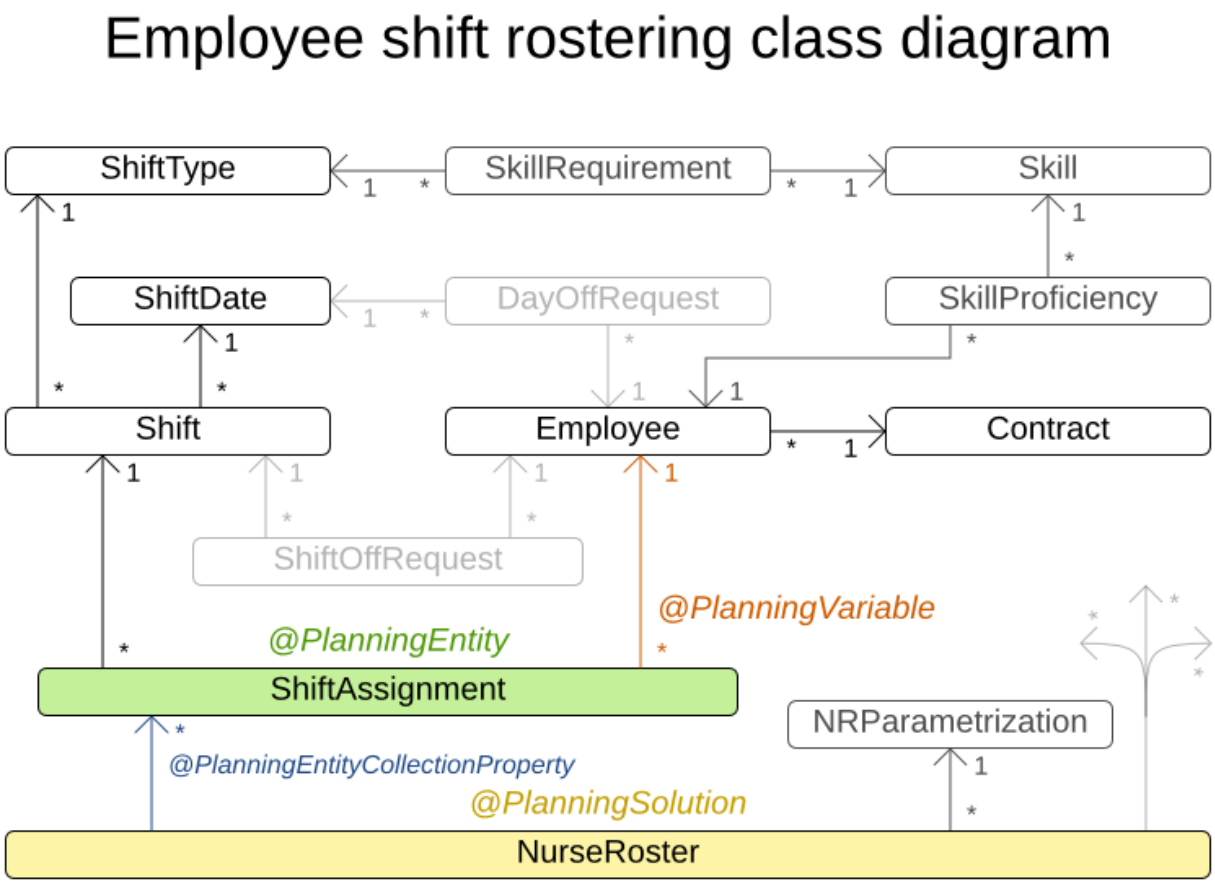
> toy1 has 1 skills, 3 shiftTypes, 2 patterns, 1 contracts, 6 employees, 7 shiftDates, 35 shiftAssignments and 0 requests with a search space of 10^27.
> toy2 has 1 skills, 3 shiftTypes, 3 patterns, 2 contracts, 20 employees, 28 shiftDates, 180 shiftAssignments and 140 requests with a search space of 10^234.
>
> sprint01 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
> sprint02 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
> sprint03 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
> sprint04 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
> sprint05 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.

sprint06 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint07 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint08 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint09 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint10 has 1 skills, 4 shiftTypes, 3 patterns, 4 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_hint01 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_hint02 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_hint03 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_late01 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_late02 has 1 skills, 3 shiftTypes, 4 patterns, 3 contracts, 10 employees, 28 shiftDates, 144 shiftAssignments and 139 requests with a search space of 10^144.
sprint_late03 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 160 shiftAssignments and 150 requests with a search space of 10^160.
sprint_late04 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 160 shiftAssignments and 150 requests with a search space of 10^160.
sprint_late05 has 1 skills, 4 shiftTypes, 8 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_late06 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_late07 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.
sprint_late08 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 0 requests with a search space of 10^152.
sprint_late09 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 0 requests with a search space of 10^152.
sprint_late10 has 1 skills, 4 shiftTypes, 0 patterns, 3 contracts, 10 employees, 28 shiftDates, 152 shiftAssignments and 150 requests with a search space of 10^152.

medium01 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28 shiftDates, 608 shiftAssignments and 403 requests with a search space of 10^906.
medium02 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28 shiftDates, 608 shiftAssignments and 403 requests with a search space of 10^906.
medium03 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28 shiftDates, 608 shiftAssignments and 403 requests with a search space of 10^906.
medium04 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28 shiftDates, 608 shiftAssignments and 403 requests with a search space of 10^906.
medium05 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 31 employees, 28 shiftDates, 608 shiftAssignments and 403 requests with a search space of 10^906.
medium_hint01 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees, 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of 10^632.
medium_hint02 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees, 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of 10^632.
medium_hint03 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees, 28 shiftDates, 428 shiftAssignments and 390 requests with a search space of 10^632.
medium_late01 has 1 skills, 4 shiftTypes, 7 patterns, 4 contracts, 30 employees, 28 shiftDates, 424 shiftAssignments and 390 requests with a search space of 10^626.
medium_late02 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees, 28 shiftDates, 428

```
shiftAssignments and 390 requests with a search space of  10^632.
medium_late03 has 1 skills, 4 shiftTypes, 0 patterns, 4 contracts, 30 employees, 28 shiftDates, 428
shiftAssignments and 390 requests with a search space of  10^632.
medium_late04 has 1 skills, 4 shiftTypes, 7 patterns, 3 contracts, 30 employees, 28 shiftDates, 416
shiftAssignments and 390 requests with a search space of  10^614.
medium_late05 has 2 skills, 5 shiftTypes, 7 patterns, 4 contracts, 30 employees, 28 shiftDates, 452
shiftAssignments and 390 requests with a search space of  10^667.

long01        has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees, 28 shiftDates, 740
shiftAssignments and 735 requests with a search space of 10^1250.
long02        has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees, 28 shiftDates, 740
shiftAssignments and 735 requests with a search space of 10^1250.
long03        has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees, 28 shiftDates, 740
shiftAssignments and 735 requests with a search space of 10^1250.
long04        has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees, 28 shiftDates, 740
shiftAssignments and 735 requests with a search space of 10^1250.
long05        has 2 skills, 5 shiftTypes, 3 patterns, 3 contracts, 49 employees, 28 shiftDates, 740
shiftAssignments and 735 requests with a search space of 10^1250.
long_hint01   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees, 28 shiftDates, 740
shiftAssignments and   0 requests with a search space of 10^1257.
long_hint02   has 2 skills, 5 shiftTypes, 7 patterns, 3 contracts, 50 employees, 28 shiftDates, 740
shiftAssignments and   0 requests with a search space of 10^1257.
long_hint03   has 2 skills, 5 shiftTypes, 7 patterns, 3 contracts, 50 employees, 28 shiftDates, 740
shiftAssignments and   0 requests with a search space of 10^1257.
long_late01   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees, 28 shiftDates, 752
shiftAssignments and   0 requests with a search space of 10^1277.
long_late02   has 2 skills, 5 shiftTypes, 9 patterns, 4 contracts, 50 employees, 28 shiftDates, 752
shiftAssignments and   0 requests with a search space of 10^1277.
long_late03   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees, 28 shiftDates, 752
shiftAssignments and   0 requests with a search space of 10^1277.
long_late04   has 2 skills, 5 shiftTypes, 9 patterns, 4 contracts, 50 employees, 28 shiftDates, 752
shiftAssignments and   0 requests with a search space of 10^1277.
long_late05   has 2 skills, 5 shiftTypes, 9 patterns, 3 contracts, 50 employees, 28 shiftDates, 740
shiftAssignments and   0 requests with a search space of 10^1257.
```

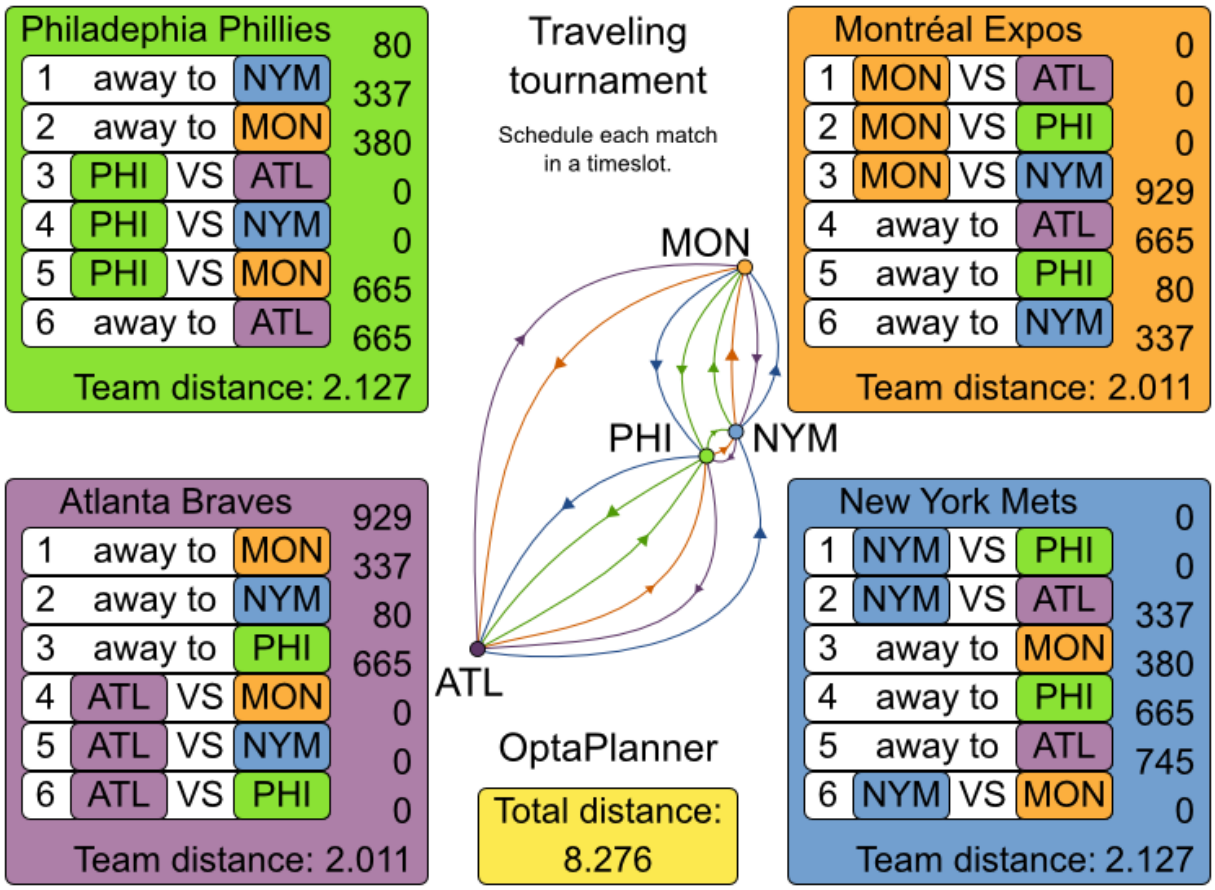
3.4.2.3. ドメインモデル



3.4.3. 巡回トーナメント問題 (TTP)

3.4.3.1. 問題の詳細

n チーム間の試合をスケジュールします。



ハード制約:

- 各チームは、他のチームとそれぞれ 2 回 (ホームとアウェイ) 試合をする。
- 各チームは、各時間枠に 1 試合だけ行う。
- 3 回連続で、ホームまたはアウェイでの試合はできない。
- 繰り返しなし: 同じ対戦相手と 2 回連続で対戦できない。

ソフト制約:

- 全チームが移動する合計距離を最小限に抑える。

この問題は Michael Trick の Web サイト (世界記録が含まれます) で定義されています。

3.4.3.2. 問題の規模

```
1-nl04    has  6 days,  4 teams and   12 matches with a search space of    10^9.
1-nl06    has 10 days,  6 teams and   30 matches with a search space of   10^30.
1-nl08    has 14 days,  8 teams and   56 matches with a search space of   10^64.
1-nl10    has 18 days, 10 teams and   90 matches with a search space of  10^112.
1-nl12    has 22 days, 12 teams and  132 matches with a search space of  10^177.
1-nl14    has 26 days, 14 teams and  182 matches with a search space of  10^257.
1-nl16    has 30 days, 16 teams and  240 matches with a search space of  10^354.
2-bra24    has 46 days, 24 teams and  552 matches with a search space of  10^917.
3-nfl16   has 30 days, 16 teams and  240 matches with a search space of  10^354.
3-nfl18   has 34 days, 18 teams and  306 matches with a search space of  10^468.
```

```
3-nfl20    has 38 days, 20 teams and  380 matches with a search space of  10^600.
3-nfl22    has 42 days, 22 teams and  462 matches with a search space of  10^749.
3-nfl24    has 46 days, 24 teams and  552 matches with a search space of  10^917.
3-nfl26    has 50 days, 26 teams and  650 matches with a search space of 10^1104.
3-nfl28    has 54 days, 28 teams and  756 matches with a search space of 10^1309.
3-nfl30    has 58 days, 30 teams and  870 matches with a search space of 10^1534.
3-nfl32    has 62 days, 32 teams and  992 matches with a search space of 10^1778.
4-super04  has  6 days,  4 teams and   12 matches with a search space of    10^9.
4-super06  has 10 days,  6 teams and   30 matches with a search space of   10^30.
4-super08  has 14 days,  8 teams and   56 matches with a search space of   10^64.
4-super10  has 18 days, 10 teams and   90 matches with a search space of  10^112.
4-super12  has 22 days, 12 teams and  132 matches with a search space of  10^177.
4-super14  has 26 days, 14 teams and  182 matches with a search space of  10^257.
5-galaxy04 has  6 days,  4 teams and   12 matches with a search space of    10^9.
5-galaxy06 has 10 days,  6 teams and   30 matches with a search space of   10^30.
5-galaxy08 has 14 days,  8 teams and   56 matches with a search space of   10^64.
5-galaxy10 has 18 days, 10 teams and   90 matches with a search space of  10^112.
5-galaxy12 has 22 days, 12 teams and  132 matches with a search space of  10^177.
5-galaxy14 has 26 days, 14 teams and  182 matches with a search space of  10^257.
5-galaxy16 has 30 days, 16 teams and  240 matches with a search space of  10^354.
5-galaxy18 has 34 days, 18 teams and  306 matches with a search space of  10^468.
5-galaxy20 has 38 days, 20 teams and  380 matches with a search space of  10^600.
5-galaxy22 has 42 days, 22 teams and  462 matches with a search space of  10^749.
5-galaxy24 has 46 days, 24 teams and  552 matches with a search space of  10^917.
5-galaxy26 has 50 days, 26 teams and  650 matches with a search space of 10^1104.
5-galaxy28 has 54 days, 28 teams and  756 matches with a search space of 10^1309.
5-galaxy30 has 58 days, 30 teams and  870 matches with a search space of 10^1534.
5-galaxy32 has 62 days, 32 teams and  992 matches with a search space of 10^1778.
5-galaxy34 has 66 days, 34 teams and 1122 matches with a search space of 10^2041.
5-galaxy36 has 70 days, 36 teams and 1260 matches with a search space of 10^2324.
5-galaxy38 has 74 days, 38 teams and 1406 matches with a search space of 10^2628.
5-galaxy40 has 78 days, 40 teams and 1560 matches with a search space of 10^2951.
```

3.4.4. コストを抑えるスケジュール

3.4.4.1. 問題の詳細

全タスクを時間内にスケジュールし、機械の電気代を最小限に抑えます。電気代は時間によって異なります。これは、ジョブショップスケジューリング の応用です。

ハード制約:

- 開始時間の制限: 各タスクは、最早と最遅の開始時間の制限内に、開始する必要がある。
- 最大容量: 各機械に割り当てる各リソースは最大容量を超過することはできない。
- 開始および終了: 各機械は、タスクが割り当てられている間は稼働している必要がある。次のタスクまでの間、起動および終了コストを避けるため、機械をアイドルにすることができる。

中程度の制約:

- 電気代: 全スケジュールの合計電気代を最小限に抑える。
  - 機械の電気代: 稼働中またはアイドル中の機械はそれぞれ、電気を消費し、電気代が発生する (金額は使用時の電気代によって異なる)。

- タスクの電気代: 各タスクも電気を消費し、電気代が発生する (金額は使用時の電気代によって異なる)。
- 機械の起動および終了コスト: 機械を起動または終了するたびに、追加のコストが発生する。

ソフト制約 (問題に元々設定されている定義に追加):

- 早く開始: なるべく早めにタスクを開始するようにする。

この問題は ICON challenge で定義されています。

3.4.4.2. 問題の規模

```
sample01   has 3 resources,   2 machines, 288 periods and   25 tasks with a search space of
10^53.
sample02   has 3 resources,   2 machines, 288 periods and   50 tasks with a search space of
10^114.
sample03   has 3 resources,   2 machines, 288 periods and  100 tasks with a search space of
10^226.
sample04   has 3 resources,   5 machines, 288 periods and  100 tasks with a search space of
10^266.
sample05   has 3 resources,   2 machines, 288 periods and  250 tasks with a search space of
10^584.
sample06   has 3 resources,   5 machines, 288 periods and  250 tasks with a search space of
10^673.
sample07   has 3 resources,   2 machines, 288 periods and 1000 tasks with a search space of
10^2388.
sample08   has 3 resources,   5 machines, 288 periods and 1000 tasks with a search space of
10^2748.
sample09   has 4 resources,  20 machines, 288 periods and 2000 tasks with a search space of
10^6668.
instance00 has 1 resources,  10 machines, 288 periods and  200 tasks with a search space of
10^595.
instance01 has 1 resources,  10 machines, 288 periods and  200 tasks with a search space of
10^599.
instance02 has 1 resources,  10 machines, 288 periods and  200 tasks with a search space of
10^599.
instance03 has 1 resources,  10 machines, 288 periods and  200 tasks with a search space of
10^591.
instance04 has 1 resources,  10 machines, 288 periods and  200 tasks with a search space of
10^590.
instance05 has 2 resources,  25 machines, 288 periods and  200 tasks with a search space of
10^667.
instance06 has 2 resources,  25 machines, 288 periods and  200 tasks with a search space of
10^660.
instance07 has 2 resources,  25 machines, 288 periods and  200 tasks with a search space of
10^662.
instance08 has 2 resources,  25 machines, 288 periods and  200 tasks with a search space of
10^651.
instance09 has 2 resources,  25 machines, 288 periods and  200 tasks with a search space of
10^659.
instance10 has 2 resources,  20 machines, 288 periods and  500 tasks with a search space of
10^1657.
instance11 has 2 resources,  20 machines, 288 periods and  500 tasks with a search space of
10^1644.
```

instance12 has 2 resources, 20 machines, 288 periods and 500 tasks with a search space of 10^1637.
instance13 has 2 resources, 20 machines, 288 periods and 500 tasks with a search space of 10^1659.
instance14 has 2 resources, 20 machines, 288 periods and 500 tasks with a search space of 10^1643.
instance15 has 3 resources, 40 machines, 288 periods and 500 tasks with a search space of 10^1782.
instance16 has 3 resources, 40 machines, 288 periods and 500 tasks with a search space of 10^1778.
instance17 has 3 resources, 40 machines, 288 periods and 500 tasks with a search space of 10^1764.
instance18 has 3 resources, 40 machines, 288 periods and 500 tasks with a search space of 10^1769.
instance19 has 3 resources, 40 machines, 288 periods and 500 tasks with a search space of 10^1778.
instance20 has 3 resources, 50 machines, 288 periods and 1000 tasks with a search space of 10^3689.
instance21 has 3 resources, 50 machines, 288 periods and 1000 tasks with a search space of 10^3678.
instance22 has 3 resources, 50 machines, 288 periods and 1000 tasks with a search space of 10^3706.
instance23 has 3 resources, 50 machines, 288 periods and 1000 tasks with a search space of 10^3676.
instance24 has 3 resources, 50 machines, 288 periods and 1000 tasks with a search space of 10^3681.
instance25 has 3 resources, 60 machines, 288 periods and 1000 tasks with a search space of 10^3774.
instance26 has 3 resources, 60 machines, 288 periods and 1000 tasks with a search space of 10^3737.
instance27 has 3 resources, 60 machines, 288 periods and 1000 tasks with a search space of 10^3744.
instance28 has 3 resources, 60 machines, 288 periods and 1000 tasks with a search space of 10^3731.
instance29 has 3 resources, 60 machines, 288 periods and 1000 tasks with a search space of 10^3746.
instance30 has 4 resources, 70 machines, 288 periods and 2000 tasks with a search space of 10^7718.
instance31 has 4 resources, 70 machines, 288 periods and 2000 tasks with a search space of 10^7740.
instance32 has 4 resources, 70 machines, 288 periods and 2000 tasks with a search space of 10^7686.
instance33 has 4 resources, 70 machines, 288 periods and 2000 tasks with a search space of 10^7672.
instance34 has 4 resources, 70 machines, 288 periods and 2000 tasks with a search space of 10^7695.
instance35 has 4 resources, 80 machines, 288 periods and 2000 tasks with a search space of 10^7807.
instance36 has 4 resources, 80 machines, 288 periods and 2000 tasks with a search space of 10^7814.
instance37 has 4 resources, 80 machines, 288 periods and 2000 tasks with a search space of 10^7764.
instance38 has 4 resources, 80 machines, 288 periods and 2000 tasks with a search space of 10^7736.
instance39 has 4 resources, 80 machines, 288 periods and 2000 tasks with a search space of 10^7783.

> instance40 has 4 resources, 90 machines, 288 periods and 4000 tasks with a search space of 10^15976.
> instance41 has 4 resources, 90 machines, 288 periods and 4000 tasks with a search space of 10^15935.
> instance42 has 4 resources, 90 machines, 288 periods and 4000 tasks with a search space of 10^15887.
> instance43 has 4 resources, 90 machines, 288 periods and 4000 tasks with a search space of 10^15896.
> instance44 has 4 resources, 90 machines, 288 periods and 4000 tasks with a search space of 10^15885.
> instance45 has 4 resources, 100 machines, 288 periods and 5000 tasks with a search space of 10^20173.
> instance46 has 4 resources, 100 machines, 288 periods and 5000 tasks with a search space of 10^20132.
> instance47 has 4 resources, 100 machines, 288 periods and 5000 tasks with a search space of 10^20126.
> instance48 has 4 resources, 100 machines, 288 periods and 5000 tasks with a search space of 10^20110.
> instance49 has 4 resources, 100 machines, 288 periods and 5000 tasks with a search space of 10^20078.

3.4.5. 投資資産クラスの割り当て (ポートフォリオの最適化)

3.4.5.1. 問題の詳細

各資産クラスに投資する相対数を決定します。

ハード制約:

- リスクの最大値: 標準偏差合計は、標準偏差の最大値を超えてはならない。
    - 標準偏差合計の計算は、Markowitz Portfolio Theory を適用した、資産クラスの相対関係を考慮する必要があります。
- 地域の最大値: 地域ごとに数量の最大値がある。
- セクターの最大値: 各セクターに数量の最大値がある。

ソフト制約:

- 期待収益を最大化する。

3.4.5.2. 問題の規模

> de_smet_1 has 1 regions, 3 sectors and 11 asset classes with a search space of 10^4.
> irrinki_1 has 2 regions, 3 sectors and 6 asset classes with a search space of 10^3.

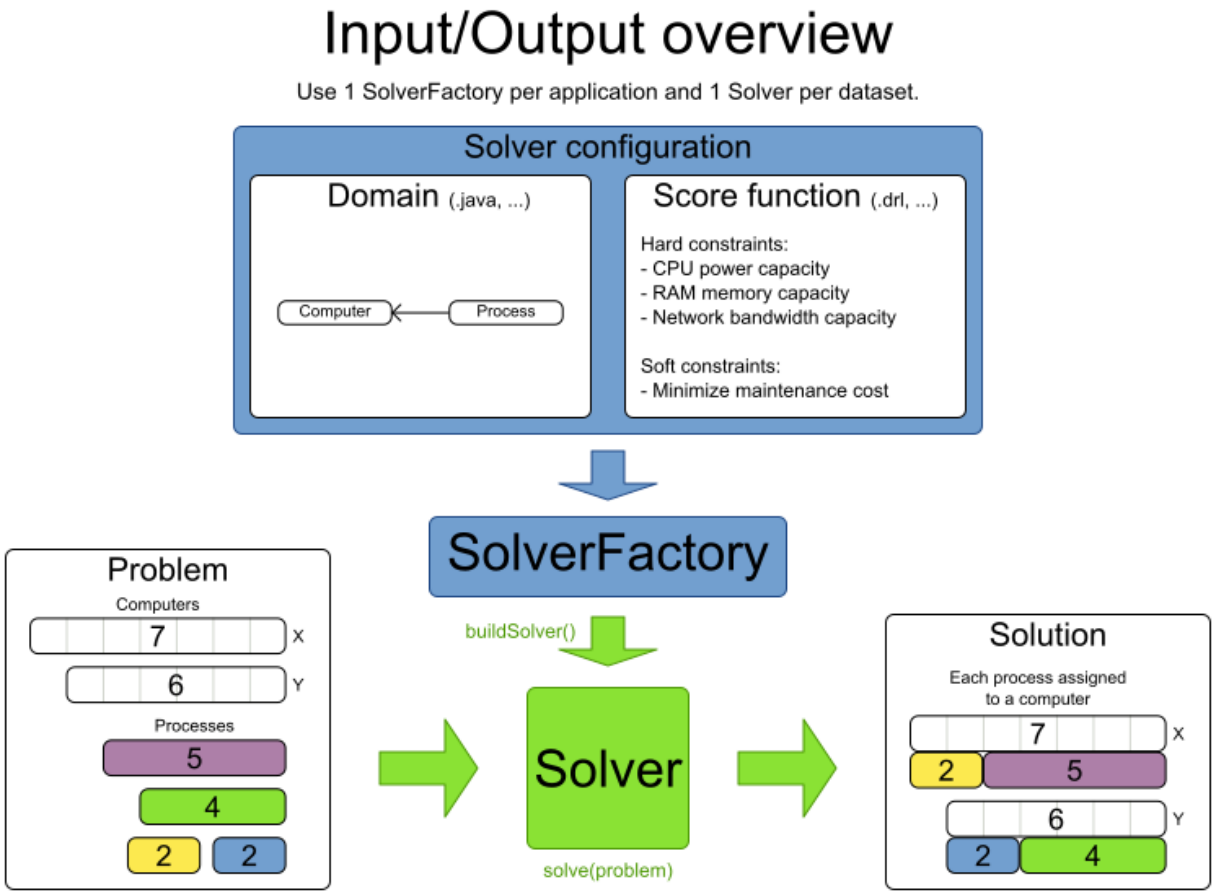
サイズが大きいデータセットは作成/検証されていませんが、問題はないはずです。

第4章 PLANNER の設定

4.1. 概要

以下の 4 つの手順を行い、Planner で計画問題を解決します。

1. 計画問題を、**Solution** インターフェースを実装するクラス (**NQueens** クラスなど) として モデル化します。
2. **Solver** を設定します (**NQueens** インスタンスの FF (First Fit) やタブー探索など)。
3. データ層から 問題のデータセットを読み込みます (クィーン 4 個の例など)。これが計画問題です。
4. 見つけた最適解を返す **Solver.solve(planningProblem)** で、問題を解決します。



4.2. SOLVER の設定

4.2.1. XML で Solver の設定

SolverFactory で **Solver** インスタンスを作成します。クラスパスリソースとして提供されている XML の Solver 設定ファイルで **SolverFactory** を設定します (**ClassLoader.getResource()** で定義)。

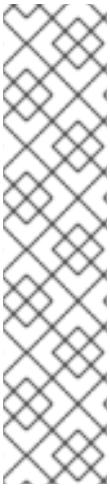
```
SolverFactory<NQueens> solverFactory = SolverFactory.createFromXmlResource(
        "org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml");
Solver<NQueens> solver = solverFactory.buildSolver();
```

通常のプロジェクト (Maven ディレクトリーの構造に準拠) では、XML の solverConfig ファイルは **$PROJECT_DIR/src/main/resources/org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml** にあります。または、**SolverFactory.createFromXmlFile()** などのメソッドを使用して、**File**、**InputStream**、または **Reader** から **SolverFactory** を作成することもできます。ただし、移植性の理由から、クラスパスリソースが推奨されます。

注記

一部の環境 (OSGi、JBoss modules など) では、JAR のクラスパス (Solver 設定、スコアの DRL およびドメインクラス) は、**optaplanner-core** JAR のデフォルトの **ClassLoader** では利用できない可能性があります。このような場合には、以下のように、クラスの **ClassLoader** をパラメーターとして指定します。

```
SolverFactory<NQueens> solverFactory =
SolverFactory.createFromXmlResource(
        ".../nqueensSolverConfig.xml", getClass().getClassLoader());
```



注記

Workbench または Execution Server を使用する場合に、Drools の **KieContainer** 機能を活用するには、**KieContainer** をパラメーターとして指定します。

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.newKieContainer(
        kieServices.newReleaseId("org.nqueens", "nqueens", "1.0.0"));
SolverFactory<NQueens> solverFactory =
SolverFactory.createFromKieContainerXmlResource(
        kieContainer, ".../nqueensSolverConfig.xml");
```

Solver 設定の ksessionName を使用します。

Solver および **SolverFactory** にはいずれも、問題の計画と解 (ソリューション) を表現するクラスである **Solution_** という汎用型が含まれています。

XML の Solver 設定ファイルは以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<solver>
  <!-- Define the model -->
  <solutionClass>org.optaplanner.examples.nqueens.domain.NQueens</solutionClass>
  <entityClass>org.optaplanner.examples.nqueens.domain.Queen</entityClass>

  <!-- Define the score function -->
  <scoreDirectorFactory>
    <scoreDefinitionType>SIMPLE</scoreDefinitionType>
    <scoreDrl>org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</scoreDrl>
  </scoreDirectorFactory>

  <!-- Configure the optimization algorithms (optional) -->
```

```
  <termination>
    ...
  </termination>
  <constructionHeuristic>
    ...
  </constructionHeuristic>
  <localSearch>
    ...
  </localSearch>
</solver>
```

これは、以下の 3 つの部分で構成されています。

- モデルの定義
- スコア関数の定義
- 最適化アルゴリズムの定義 (任意)

本書では、この設定について、詳しく説明しています。

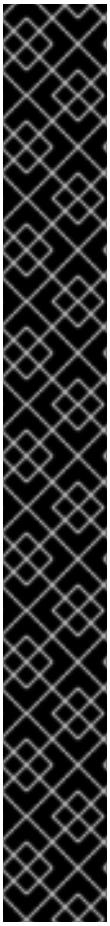
Planner では、設定を変更するだけで最適化アルゴリズムを比較的簡単に切り替えることができます。異なる設定を試して、各ユースケースに最適な設定を報告できる ベンチマーカー もあります。

4.2.2. Java API で Solver の設定

Solver は、**SolverConfig** API を使用して設定することもできます。これは、ランタイム時に動的に値を変更する場合に特に便利です。たとえば、**Solver** を構築する前に、ユーザーの入力に合わせて実行時間を変更するには、以下のように設定します。

```
SolverFactory<NQueens> solverFactory = SolverFactory.createFromXmlResource(
        "org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml");

TerminationConfig terminationConfig = new TerminationConfig();
terminationConfig.setMinutesSpentLimit(userInput);
solverFactory.getSolverConfig().setTerminationConfig(terminationConfig);

Solver<NQueens> solver = solverFactory.buildSolver();
```

XML の Solver 設定の全要素は、***Config** クラス、またはパッケージ名前空間 **org.optaplanner.core.config** の ***Config** クラスにあるプロパティーとして利用できます。この ***Config** クラスは、XML 形式の Java 表現で、(パッケージ名前空間 **org.optaplanner.core.impl** の) ランタイムコンポーネントを構築して効率的な **Solver** を組み立てます。



重要

SolverFactory は、設定しないとマルチスレッドセーフにならないため、**getSolverConfig()** メソッドはスレッドセーフではありません。ユーザー要求に合わせて動的に **SolverFactory** を設定するには、初期化時にベースとして **SolverFactory** を構築し、ユーザー要求に対して、**cloneSolverFactory()** メソッドでクローンを作成します。

```
    private SolverFactory<NQueens> base;

    public void init() {
        base = SolverFactory.createFromXmlResource(
                "org/optaplanner/examples/nqueens/solver/nqueensSolverConfig.xml");
        base.getSolverConfig().setTerminationConfig(new TerminationConfig());
    }

    // Called concurrently from different threads
    public void userRequest(..., long userInput)
        SolverFactory<NQueens> solverFactory = base.cloneSolverFactory();

solverFactory.getSolverConfig().getTerminationConfig().setMinutesSpentLimit(userInput
);
        Solver<NQueens> solver = solverFactory.buildSolver();
        ...
    }
```

4.2.3. Business Central での Solver 設定

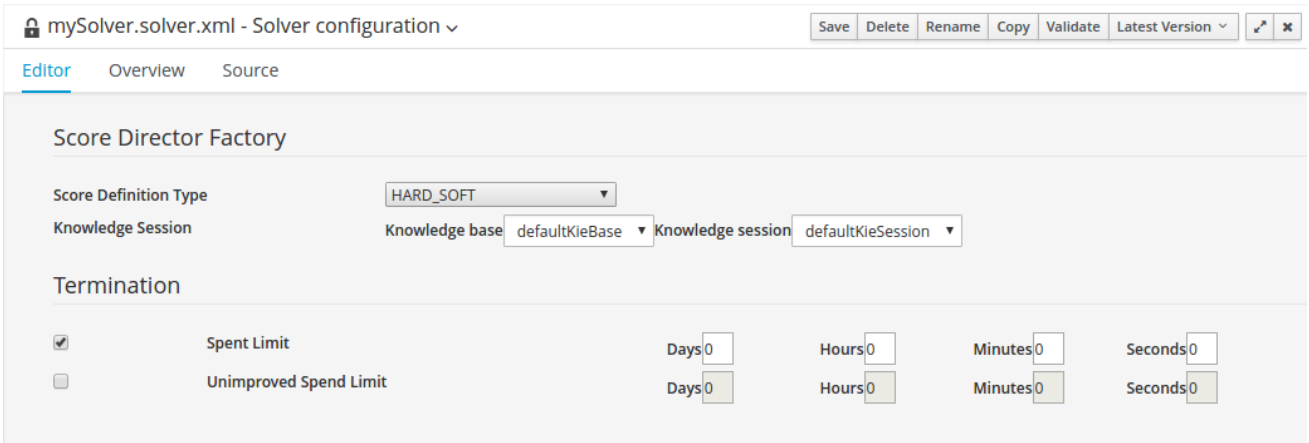
Solver の設定には、Business Central の Solver エディターを使用します。Business Central に関する情報は、『Red Hat JBoss BPM Suite User Guide』の 「1.4. 章 Business Central」を参照してください。

注記

Business Central の Business Resource Planner 設定を有効にするには、**plannermgmt** ロールが必要です。

Business Central に Solver エディターを作成するには、**New Item → Solver configuration** をクリックします。DRL ルールやガイド付きのデシジョンテーブルなど、他のアセットを使用することもできる点に注意してください。

図4.1 Solver 設定エディター



Solver エディターは、基本的な Solver 設定を作成します。KJAR のデプロイ後、Realtime Decision Server またはプレーンな Java コードで、DRL ルール、Planner エンティティー、Planner ソリューションと合わせて実行できます。

エディターで Solver を定義したら、**Source** をクリックして、XML 形式で表示します。

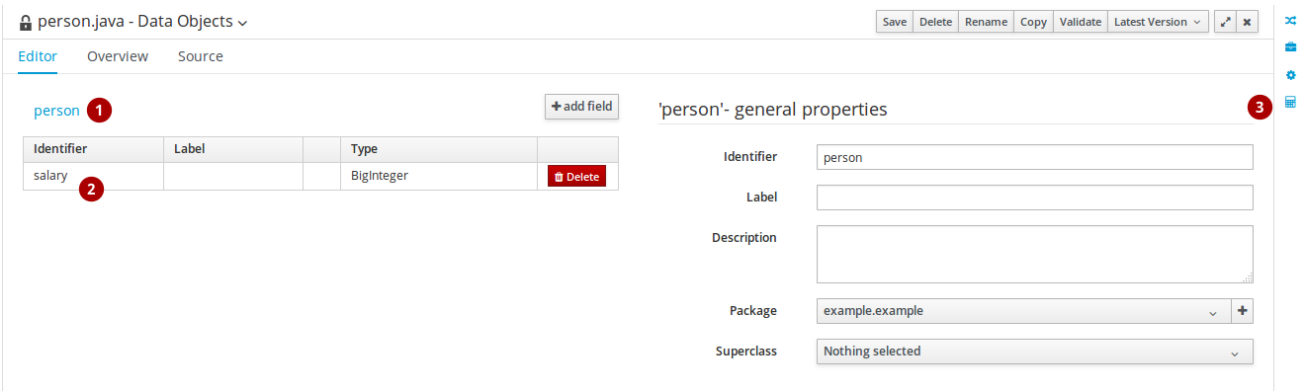
```
<solver xStreamId="1">
  <scanAnnotatedClasses xStreamId="2"/>
  <scoreDirectorFactory xStreamId="3">
    <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  </scoreDirectorFactory>
  <termination xStreamId="4">
    <secondsSpentLimit>0</secondsSpentLimit>
    <minutesSpentLimit>0</minutesSpentLimit>
    <hoursSpentLimit>5</hoursSpentLimit>
    <daysSpentLimit>0</daysSpentLimit>
  </termination>
</solver>
```

Validate ボタンを使用して、Solver 設定を検証します。これにより Solver が作成され、デプロイや実行を行うことなく、プロジェクトの問題が多数提示されます。

デフォルトでは、Solver の設定は、プランニングエンティティーとプランニングソリューションのクラスをすべて自動的にスキャンします。何も見つからない (もしくは結果が多すぎる) と、検証に失敗します。

Data Object タブ
Business Resource Planner で使用するプランニングエンティティーとプランニングソリューションをモデル化するには、データオブジェクトを使用します。データオブジェクトに関する情報は、『Red Hat JBoss BPM Suite User Guide』の「4.14 章 Data models」を参照してください。データオブジェクトを、Planner エンティティーまたはソリューションとして指定する方法は、以下の手順を参照してください。

図4.2 Data Object タブ



データオブジェクトのラベルをクリックし、OptaPlanner ツールウィンドウで、オブジェクトを、プランニングエンティティーまたはプランニングソリューションとして設定します。

特定のデータオブジェクトフィールドをクリックして、OptaPlanner ツールウィンドウで Business Resource Planner 変数および Solver の関係を設定します。

OptaPlanner ツールウィンドウでは、コンテキストをもとに Planner 設定を変更できます。つまり、選択した内容によって、このウィンドウの内容が変化します。

4.2.4. アノテーション設定

4.2.4.1. アノテーションの自動スキャン

@PlanningSolution または **@PlanningEntity** が含まれるクラスを手動で宣言する代わりに以下を行います。

```
<solver>
  <!-- Define the model -->
  <solutionClass>org.optaplanner.examples.nqueens.domain.NQueens</solutionClass>
  <entityClass>org.optaplanner.examples.nqueens.domain.Queen</entityClass>

  ...
</solver>
```

Planner では、自動的にクラスパスをスキャンして、検索します。

```
<solver>
  <!-- Define the model -->
  <scanAnnotatedClasses/>

  ...
</solver>
```

クラスパスに複数のモデルがある場合 (またはスキャンのスピードを速める場合) は、以下のようにスキャンするパッケージを指定します。

```
<solver>
  <!-- Define the model -->
  <scanAnnotatedClasses>
    <packageInclude>org.optaplanner.examples.cloudbalancing</packageInclude>
  </scanAnnotatedClasses>

  ...
</solver>
```

これにより、パッケージまたはサブパッケージのソリューションおよびエンティティークラスがすべて検出されます。

注記

scanAnnotatedClasses を指定していない場合は、**org.reflections** の Maven 遷移的依存関係を除外することができます。

4.2.4.2. その他のアノテーション方法

Planner は、ドメインモデルのどのクラスが、プランニング変数などをプロパティーとするプランニングエンティティーであるかを指定する必要があります。この情報を渡す方法は複数あります。

- ドメインモデルにクラスアノテーションと JavaBean プロパティーアノテーションを追加します (推奨)。プロパティーアノテーションは、セッターメソッドではなく、ゲッターメソッドに指定する必要があります。このゲッターメソッドを公開する必要はありません。

- ドメインモデルにクラスアノテーションとフィールドアノテーションを追加します。このフィールドは公開する必要がありません。
- アノテーションなし: XML ファイルでのドメイン設定を外部に設置します。これは、未対応 です。

本書は、1つ目の方法に焦点を当てていますが、明示的に記してない場合でも、すべての機能でこの3つの方法に対応しています。

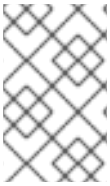
4.3. 計画問題のモデル化

4.3.1. 問題ファクトクラスまたはプランニングエンティティークラス

計画問題のデータセットについて見ていきます。このデータセットに、ドメインクラスがあることが分かるでしょう。これらのドメインクラスは、以下のいずれかに分類できます。

- 関連性のないクラス: どのスコア制約でも使用されません。計画に関して言えば、このデータは使用されません。
- 問題ファクト クラス: スコア制約により使用されますが、(問題が変わらない限り)、計画時に変化しません (例: **Bed**、**Room**、**Shift**、**Employee**、**Topic**、**Period** など)。問題ファクトクラスのプロパティーはすべて、問題プロパティーです。
- プランニングエンティティー クラス: スコア制約により使用され、計画時に変化します (**BedDesignation**、**ShiftAssignment**、**Exam** など)。計画時に変化するプロパティーはプランニング変数で、他のプロパティーは問題プロパティーです。

計画時に変化するクラスと、 **Solver** を使用して変更する変数が含まれるクラス を確認してください。そのクラスがプランニングエンティティーです。多くのユースケースでは、1つのプランニングエンティティークラスにプランニング変数が1つだけ含まれます。



注記

リアルタイムの計画 では、問題自体が変化しても、計画時に問題ファクトは変化せず、(Solver は一時的に停止して問題ファクトの変更を適用するため) 次回の計画までに変化します。

適切なモデルを使用すれば、計画の実装が成功する可能性が大幅に高まります。以下のガイドラインに従い、適切なモデルを設計してください。

- 多対一 の関係に当てはめると、通常、プランニングエンティティークラスが 多 になり、他方を参照するプロパティーはプランニング変数ということになります。「従業員の勤務表」サンプルでは、プランニングエンティティークラスは **ShiftAssignment** で、**Employee** ではありません。プランニング変数は **ShiftAssignment.getEmployee()** です。なぜなら、**ShiftAssignments** は、1つの **Employee** に対して複数割り当てられますが、1つの **ShiftAssignment** には **Employee** が1つしか割り当てられないためです。
- プランニングエンティティークラスには、最低でも問題プロパティーが1つ必要です。プランニング変数だけを持つプランニングエンティティーは、通常、プランニング変数の1つを問題プロパティーに変換して簡素化することができます。これにより、探索空間のサイズ が大幅に縮小されます。たとえば、「従業員の勤務表」では、**ShiftAssignment** の **getShift()** が問題プロパティーで、**getEmployee()** はプランニング変数です。両方をプランニング変数にすると、問題を解決する際の効率がはるかに悪くなります。
  - 問題プロパティーは最低でも1つ必要になりますが、代わりに ID を使用するのは十分では

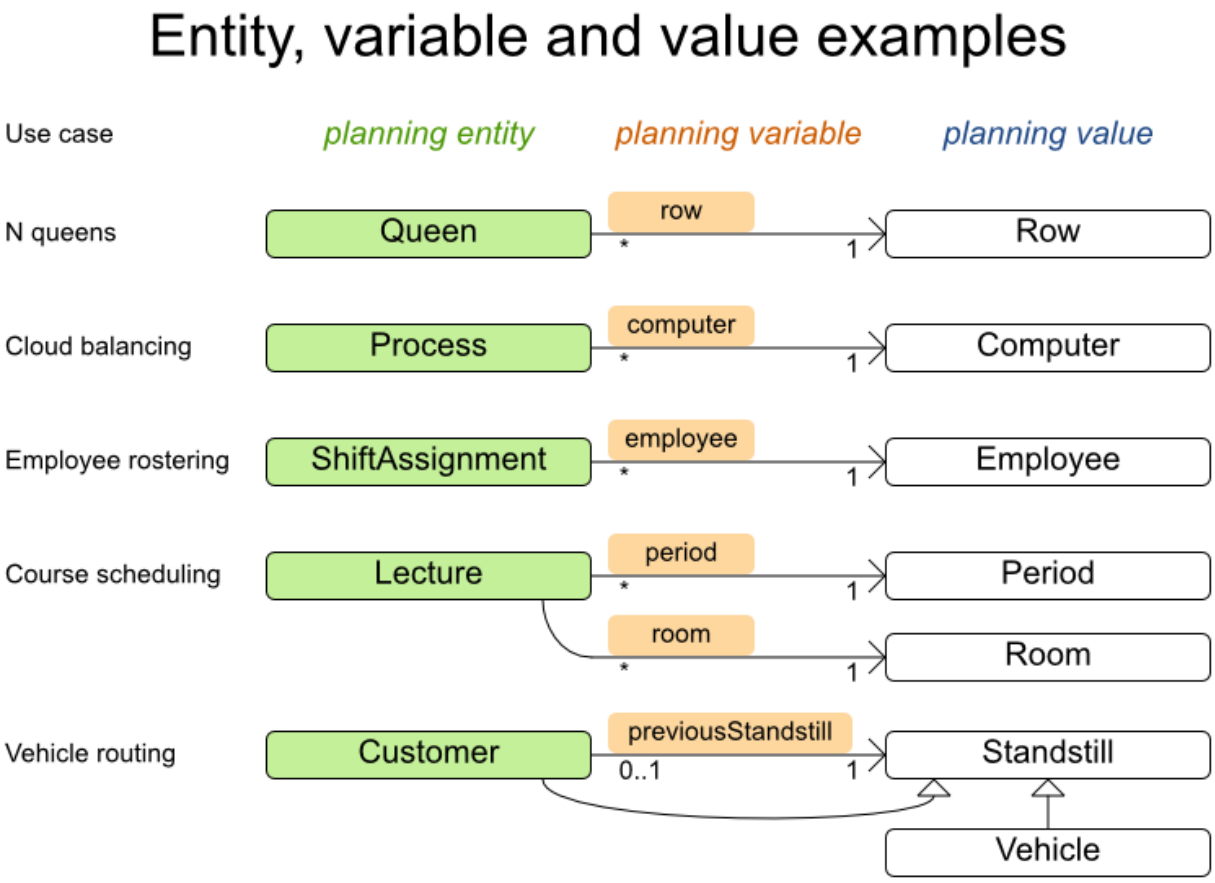
ありません。ビジネスでも理解できる必要があるため、ビジネスキーがあれば十分です。これにより、未割り当てのエンティティーが無名になることを回避できます (ビジネス側で特定できない状態を回避します)。

- こうすることで、プランニングエンティティーを 2 つにするハード制約を追加する必要がなくなります。このプランニングエンティティーは、それぞれの問題プロパティーにより異なっているためです。

- 場合によっては、複数のプランニングエンティティーで問題プロパティーが同じになる可能性があります。このような場合には、追加の問題プロパティーを作成して区別すると便利です。「従業員の勤務表」サンプルの **ShiftAssignment** には、**Shift** の問題プロパティーと、**indexInShift** の問題プロパティーがあります。

- 計画時には、プランニングエンティティーの数を変更しないことが推奨されます。どのプロパティーがプランニング変数で、どのプロパティーが問題プロパティーにすべきか不明な場合には、エンティティーの数が一定になるように選択してください。たとえば、「従業員の勤務表」サンプルで、プランニングエンティティークラス **EmployeeAssignment** に、問題プロパティー **getEmployee()**とプランニング変数 **getShift()** がある場合、各 **Employee** に **EmployeeAssignment** インスタンスがいくつ作成されるかを正しく推測することは不可能です。

参考として、一般的な設計パターン や、これらのサンプルがドメインをモデル化する方法を確認してください。



注記

配送経路は、連鎖型のプランニング変数 を使用するため特別です。

Planner では、すべての問題ファクトおよびプランニングエンティティーは POJO (Plain Old JavaBean) です。データベース、XML ファイル、データリポジトリー、REST サービス、noSQL cloud などからこれらを読み込むため、問題はありません (「統合」を参照)。

4.3.2. 問題ファクト

問題ファクトとは、計画時に変化しないゲッターを持つ JavaBean (POJO) のことです。インターフェース **Serializable** の実装が推奨されます (が、必須ではありません)。「N クィーン」の例では、行と列が問題ファクトです。

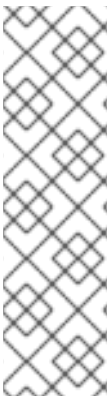
```
public class Column implements Serializable {

    private int index;

    // ... getters
}
```

```
public class Row implements Serializable {

    private int index;

    // ... getters
}
```

問題ファクトも、他の問題ファクトを参照することができます。

```
public class Course implements Serializable {

    private String code;

    private Teacher teacher; // Other problem fact
    private int lectureSize;
    private int minWorkingDaySize;

    private List<Curriculum> curriculumList; // Other problem facts
    private int studentSize;

    // ... getters
}
```

問題ファクトクラスには、Planner 固有のコードは必要ありません。たとえば、JPA アノテーションが付けられているドメインクラスを再利用することができます。



注記

一般的に、ドメインクラスがより適切に設計されていると、スコア制約がより簡単で効率的になります。そのため、整理されていない (正規化されていない) レガシーシステムを扱う場合には、最初に、整理されていないドメインモデルを Planner 固有のモデルに変換することが推奨されます。たとえば、教師が 2 つの学科に所属し、ドメインモデルの **Teacher** インスタンスが 2 つになる場合は、そのドメインモデルを調整した方が、元のモデルの空き時間に制約を課すスコア制約を記述するよりも簡単になります。

場合によっては、キャッシュされた問題ファクト を導入して、計画のみを対象とするドメインモデルを改良します。

4.3.3. プランニングエンティティー

4.3.3.1. プランニングエンティティーアノテーション

プランニングエンティティーは、問題解決時に変化する JavaBean (POJO) のことで、例としては、別の行に移動する クィーン が挙げられます。また、計画の問題には複数のプランニングエンティティーが含まれます。N クィーンの問題では、各 クィーン がプランニングエンティティーですが、プランニングエンティティークラスは 1 つのみとなっています (例: **Queen** クラス)。

プランニングエンティティークラスは、**@PlanningEntity** アノテーションを付ける必要があります。

各プランニングエンティティークラスには、1 つまたは複数の プランニング変数 を追加し、それぞれに定義 プロパティーを 1 つ以上追加する必要があります。たとえば、N クィーンでは、 クィーン は 列 で定義され、行 がプランニング変数になります。これは、問題解決時にクィーンの列を変更せず、行を変更することを示しています。

```
@PlanningEntity
public class Queen {

    private Column column;

    // Planning variables: changes during planning, between score calculations.
    private Row row;

    // ... getters and setters
}
```

プランニングエンティティークラスには、複数のプランニング変数を指定することができます。たとえば、授業 は、コース やコースのインデックス (1 つのコースには複数の授業が含まれるため) で定義されます。各 授業 は、時間帯 と 講義室 のスケジュールを組む必要があるので、プランニング変数 (時間帯と講義室) は 2 つになります。たとえば、数学のコースでは、1 週間に授業が 8 個あり、最初の授業は月曜の午前 8 時に 212 号室で開催されます。

```
@PlanningEntity
public class Lecture {

    private Course course;
    private int lectureIndexInCourse;

    // Planning variables: changes during planning, between score calculations.
    private Period period;
    private Room room;

    // ...
}
```

また、自動スキャン がない場合には、Solver 設定で以下のように各プランニングエンティティーを宣言する必要があります。

```
<solver>
  ...
  <entityClass>org.optaplanner.examples.nqueens.domain.Queen</entityClass>
  ...
</solver>
```

ユースケースによっては、複数のプランニングエンティティークラスが含まれます。たとえば、鉄道路線図に、貨物および車両を追加します。このとき、1つの貨物を輸送するのに車両を複数使用したり、1つの車両で貨物を複数輸送したりできます。このように、プランニングエンティティークラスが複数になると、ユースケース実装の複雑性が増します。

注記

不要なプランニングエンティティークラスは作成しないようにしてください。不要なクラスがあると **Move** 実装が複雑になり、スコアの計算に時間がかかります。

たとえば、授業 プランニングエンティティーが変化するのに合わせて、最新の状態に保つ必要があるので、教師の空き時間合計を格納するプランニングエンティティークラスは作成しないでください。代わりに、(シャドウ変数 として) スコア制約で空き時間を計算し、教師ごとに割り出した結果を、論理的に挿入するスコアオブジェクトに配置します。

履歴データも考慮する必要がある場合には、計画枠に収まり、 この枠を含めない 過去の割り当て合計を保持する問題ファクトを作成し (これによりプランニングエンティティーが変化しても変更されない)、スコア制約が考慮されるようにします。

4.3.3.2. プランニングエンティティーの難易度

計画がより困難なプランニングエンティティーが予測できる場合には、最適化アルゴリズムはより効率的に機能します。たとえば、ビンパッキング問題ではアイテムが大きくなればなるほど詰めるのは難しく、コースの時間割では授業に出席する学生が多くなればなるほど時間割を作成するのは困難になります。また、N クィーンでは、真ん中のクィーンの配置場所を決めるのがより困難です。

そのため、以下のように、**@PlanningEntity** アノテーションに **difficultyComparatorClass** を設定できます。

```
@PlanningEntity(difficultyComparatorClass = CloudProcessDifficultyComparator.class)
public class CloudProcess {
    // ...
}
```

```
public class CloudProcessDifficultyComparator implements Comparator<CloudProcess> {

    public int compare(CloudProcess a, CloudProcess b) {
        return new CompareToBuilder()
                .append(a.getRequiredMultiplicand(), b.getRequiredMultiplicand())
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```

または、以下のように、**Solution** からも残りの問題ファクトにアクセスできるように、**@PlanningEntity** アノテーションに **difficultyWeightFactoryClass** を設定することも可能です。

```
@PlanningEntity(difficultyWeightFactoryClass = QueenDifficultyWeightFactory.class)
public class Queen {
    // ...
}
```

詳しい情報は「ソートした選択」を参照してください。

重要

難易度は、昇順に実装する必要があります。つまり、簡単なエンティティーから困難なエンティティーの順に実装します。したがって、ビンパッキング問題の場合は、「小さいアイテム < 中サイズのアイテム < 大きいアイテム」の順になります。

多くのアルゴリズムは、より困難なエンティティーから開始しますが、これは順番が逆になっているだけです。

プランニングエンティティーの難易度を比較するのに、現在のプランニング変数の状態を使用するべきではありません。構築ヒューリスティックの間、これらの変数は **null** の可能性が高くなります。たとえば、クィーン の **row** 変数は使用するべきではありません。

4.3.4. プランニング変数

4.3.4.1. プランニング変数アノテーション

プランニング変数とは、プランニングエンティティーにある JavaBean プロパティー (つまり、ゲッターとセッター) のことです。この変数は、計画時に変化する計画値を参照します。たとえば、クィーン の **row** プロパティーはプランニング変数となります。計画時に、クィーン の **row** プロパティーが **Row** に変化した場合でも、**Row** インスタンス自体は変化しません。

プランニング変数のゲッターには、**@PlanningVariable** のアノテーションを付ける必要があります。このアノテーションは、空以外の **valueRangeProviderRefs** プロパティーを必要とします。

```
@PlanningEntity
public class Queen {
    ...

    private Row row;

    @PlanningVariable(valueRangeProviderRefs = {"rowRange"})
    public Row getRow() {
        return row;
    }

    public void setRow(Row row) {
        this.row = row;
    }

}
```

valueRangeProviderRefs プロパティーは、対象のプランニング変数で許容可能な計画値は何かを定義します。このプロパティーは、1 つまたは複数の **@ValueRangeProvider** ID を参照します。

注記

@PlanningVariable アノテーションは、@PlanningEntity アノテーションが付いたクラスに所属する必要があります。このアノテーションがないと、親クラスまたはサブクラスで無視されてしまいます。

プロパティーではなく、フィールドにアノテーション を付けることも可能です。

```
@PlanningEntity
public class Queen {
    ...

    @PlanningVariable(valueRangeProviderRefs = {"rowRange"})
    private Row row;

}
```

4.3.4.2. null 許容型のプランニング変数

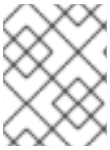
デフォルトでは、初期化されたプランニング変数は **null** にすることができないので、初期化されたソリューションが、プランニング変数に **null** を使用することはありません。過制約のユースケースではこれは逆効果です。たとえば、従業員にタスクが過剰に割り当てられてしまう場合には、優先順位の低いタスクを、負担の多いスタッフに割り当てるのではなく、未割り当ての状態にしておきます。

初期化されたプランニング変数を **null** に設定できるようにするには、**nullable** を **true** に設定します。

```
    @PlanningVariable(..., nullable = true)
    public Worker getWorker() {
        return worker;
    }
```

重要

Planner は、自動的に値の範囲に **null** 値を追加します。**ValueRangeProvider** で使用するコレクションに **null** を追加する必要はありません。



注記

null 許容型のプランニング変数を使用すると、スコア計算で null 値を使用して変数を減点 (または加点) することになります。

反復計画 (特に リアルタイム計画) は、null 許容型変数と適切に連携しません。Solver が開始したり、問題ファクトが変更したりするたびに、構築ヒューリスティック は **null** 変数すべてを再初期化しようとして、長時間無駄にしてしまう可能性があります。これに対応するには、**reinitializeVariableEntityFilter** で、プランニングエンティティーを再初期化するタイミングを変更します。

```
    @PlanningVariable(..., nullable = true, reinitializeVariableEntityFilter = ReinitializeTaskFilter.class)
    public Worker getWorker() {
        return worker;
    }
```

4.3.4.3. プランニング変数が初期化したと見なされるタイミング

プランニング変数は、値が **null** でない場合、または変数が **nullable** の場合に初期化されると見なされるため、カスタムの **reinitializeVariableEntityFilter** が構築ヒューリスティック時に再初期化をトリガーする場合でも、null 許容型変数は常に初期化すると見なされます。

プランニング変数すべてが初期化されると、プランニングエンティティーは初期化されます。

また、プランニングエンティティーがすべて初期化されると、ソリューション が初期化されます。

4.3.5. 計画値と計画値の範囲

4.3.5.1. 計画値

計画値は、プランニング変数の許容値です。通常、計画値は問題ファクトですが、**double** などのオブジェクトの場合もあります。また、別のプランニングエンティティーや、プランニングエンティティーと問題ファクトの両方が実装したインターフェースの可能性もあります。

プランニング変数の範囲は、プランニング変数が許容できる一連の値のことです。この一連の値として、可算値 (例: **1**、**2**、**3**、または **4**) または、非可算値 (例: **0.0** から **1.0** の間の **double**) などを指定できます。

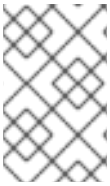
4.3.5.2. 計画値の範囲プロバイダー

4.3.5.2.1. 概要

プランニング変数の値の範囲は **@ValueRangeProvider** アノテーションで定義します。**@ValueRangeProvider** アノテーションには、**@PlanningVariable** プロパティーの **valueRangeProviderRefs** が参照する **ID** プロパティーが常に含まれます。

このアノテーションは、2 種類のメソッドに配置される可能性があります。

- ソリューションメソッド: 全プランニングエンティティーは同じ値の範囲を共有します。
- プランニングエンティティーメソッド: 値の範囲はプランニングエンティティーごとに異なり、一般的ではありません。



注記

@ValueRangeProvider アノテーションは、@PlanningSolution または @PlanningEntity アノテーションが付いたクラスに所属する必要があります。このアノテーションがないと、親クラスまたはサブクラスで無視されてしまいます。

対象のメソッドの戻り値型には 2 種類あります。

- **Collection**: 値の範囲は、許容値の **Collection** (通常 **List**) で定義されます。
- **ValueRange**: 値の範囲は、限界で定義されますが、これは一般的ではありません。

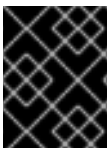
4.3.5.2.2. ソリューションの ValueRangeProvider

同じプランニングエンティティークラスの全インスタンスは、対象のプランニング変数で許容できる計画値の同じ範囲を共有します。これは、値の範囲を設定する最も一般的な方法です。

ソリューション の実装には、**Collection** (または **ValueRange**) を返すメソッドがあります。その **Collection** からの値は、対象のプランニング変数で許容できる計画値です。

```
@PlanningVariable(valueRangeProviderRefs = {"rowRange"})
public Row getRow() {
    return row;
}
```

```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {

    // ...

    @ValueRangeProvider(id = "rowRange")
    public List<Row> getRowList() {
        return rowList;
    }

}
```



重要

対象の **Collection** (または **ValueRange**) には、null 許容型プランニング変数 であっても、**null** 値を含めることはできません。

プロパティーではなく、フィールドにアノテーション を付けることも可能です。

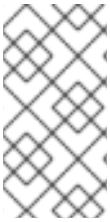
```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {
    ...

    @ValueRangeProvider(id = "rowRange")
    private List<Row> rowList;

}
```

4.3.5.2.3. プランニングエンティティーの ValueRangeProvider

プランニングエンティティーにはそれぞれ、プランニング変数に対して独自の値の範囲 (許容値のセット) があります。たとえば、教師が所属部門以外の講義室で 教鞭を取ること ができない場合に、この教師が授業を行う講義室の値の範囲が、その所属する部門の講義室に制限されてしまう可能性があります。

```
    @PlanningVariable(valueRangeProviderRefs = {"departmentRoomRange"})
    public Room getRoom() {
        return room;
    }

    @ValueRangeProvider(id = "departmentRoomRange")
    public List<Room> getPossibleRoomList() {
        return getCourse().getTeacher().getDepartment().getRoomList();
    }
```

これを使用して、ソフト制約 (または問題に実行可能解がない場合にはハード制約) を強制しないようにしてください。たとえば、所属部門以外の講義室を使う以外に、教師が授業を行うことができない場合があります。このように、所属部門以外の講義室を使う以外に手段がない可能性があるため、教師に対して講義室の値の範囲を制限すべきではありません。



注記

特定のプランニングエンティティーの値の範囲を制限することで、組み込み型のハード制約 が効率的に作成されます。これにより、可能解の数を大幅に減らす利点がありますが、最適化アルゴリズムが、一時的に制約に違反して、局所最適を回避することができなくなる可能性があります。

プランニングエンティティーは、他のプランニングエンティティーを使用して値の範囲を決定するべきではありません。他のプランニングエンティティーを使用して値の範囲を決定すると、プランニングエンティティーが問題そのものを解決しようとし、最適化アルゴリズムを妨害する結果となります。

複数のエンティティーに同じ値の範囲が割り当てられていない限り、エンティティーには独自の **List** インスタンスがあります。たとえば、教師 A および B が同じ部門に所属する場合には、同じ **List<Room>** インスタンスを使用します。さらに、各 **List** には同じ計画値のインスタンスのサブセットが含まれます。たとえば、部門 A と B の両方が講義室 X を使用できる場合に、**List<Room>** インスタンスには同じ **Room** インスタンスが含まれます。

注記

プランニングエンティティーの **ValueRangeProvider** は、ソリューションの **ValueRangeProvider** よりもメモリーを多く消費し、特定の自動パフォーマンス最適化を無効にします。

警告

プランニングエンティティーの **ValueRangeProvider** は現在、連鎖 変数には対応していません。

4.3.5.2.4. ValueRangeFactory

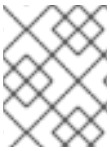
Collection の代わりに、**ValueRangeFactory** で構築される **ValueRange** または **CountableValueRange** を返すことも可能です。

```
@ValueRangeProvider(id = "delayRange")
public CountableValueRange<Integer> getDelayRange() {
    return ValueRangeFactory.createIntValueRange(0, 5000);
}
```

ValueRange は、限界のみを保持するため、使用するメモリーははるかに少なくなります。上記の例では、**Collection** には、2 つの限界だけではなく、5000 の **int** をすべて保持する必要があります。

さらに、たとえば、200 個単位で在庫を購入する必要がある場合には、**incrementUnit** を指定することができます。

```
@ValueRangeProvider(id = "stockAmountRange")
public CountableValueRange<Integer> getStockAmountRange() {
    // Range: 0, 200, 400, 600, ..., 9999600, 9999800, 10000000
    return ValueRangeFactory.createIntValueRange(0, 10000000, 200);
}
```



注記

(Planner が可算値であることが理解できるように) できる限り、**ValueRange** ではなく、**CountableValueRange** を返します

ValueRangeFactory には、複数の値クラス型に対する作成メソッドがあります。

- **int**: 32 ビットの整数の範囲
- **long**: 64 ビットの整数の範囲
- **double**: 64 ビットの浮動小数点の範囲。(**CountableValueRange** を実装しないため) 任意抽出のみをサポートします。
- **BigInteger**: 任意精度の整数の範囲
- **BigDecimal**: 小数点の範囲。デフォルトでは、インクリメントの単位は、限度の範囲内で最小となるゼロ以外の値となっています。

4.3.5.2.5. ValueRangeProviders の組み合わせ

以下のように、ValueRangeProvider は組み合わせることができます。

```
@PlanningVariable(valueRangeProviderRefs = {"companyCarRange", "personalCarRange"})
public Car getCar() {
    return car;
}
```

```
@ValueRangeProvider(id = "companyCarRange")
public List<CompanyCar> getCompanyCarList() {
    return companyCarList;
}

@ValueRangeProvider(id = "personalCarRange")
public List<PersonalCar> getPersonalCarList() {
    return personalCarList;
}
```

4.3.5.3. 計画値の強度

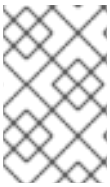
最適化アルゴリズムは、より強度の高い計画値を予測できる場合に、より効率的に機能します。つまり、プランニングエンティティーを満たす可能性が高くなります。たとえば、ビンパッキング問題では、大きい容器のほうがアイテムが入りやすく、またコースの時間割では、大きい講義室のほうが学生の収容人数の制約に違反する可能性が低くなります。

そのため、**@PlanningVariable** アノテーションに **strengthComparatorClass** を設定できます。

```
@PlanningVariable(..., strengthComparatorClass = CloudComputerStrengthComparator.class)
public CloudComputer getComputer() {
    // ...
}
```

```
public class CloudComputerStrengthComparator implements Comparator<CloudComputer> {
```

```
    public int compare(CloudComputer a, CloudComputer b) {
        return new CompareToBuilder()
                .append(a.getMultiplicand(), b.getMultiplicand())
                .append(b.getCost(), a.getCost()) // Descending (but this is debatable)
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```



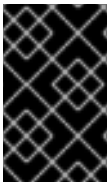
注記

同じ値の範囲に複数の計画値クラスがある場合、**strengthComparatorClass** は一般的なスーパークラスの **Comparator** を実装して (例: **Comparator<Object>**)、これらの異なるクラスのインスタンスを比較できるようにする必要があります。

または、ソリューションから残りの問題ファクトにアクセスできるように、**@PlanningVariable** アノテーションに **strengthWeightFactoryClass** を設定することも可能です。

```
@PlanningVariable(..., strengthWeightFactoryClass = RowStrengthWeightFactory.class)
public Row getRow() {
    // ...
}
```

詳しい情報は「ソートした選択」を参照してください。



重要

強度は昇順に実装する必要があります。つまり、順番は、弱いものから強いものになります。ビンパッキング問題の場合は、「小さい容器 < 中程度の容器 < 大きい容器」の順になります。

プランニングエンティティーの現在のプランニング変数を使用して、計画値を比較しないでください。構築ヒューリスティックの間、これらの変数は **null** の可能性が高くなります。たとえば、クィーン の **row** 変数を使用して、**Row** の強度を判断することはできません。

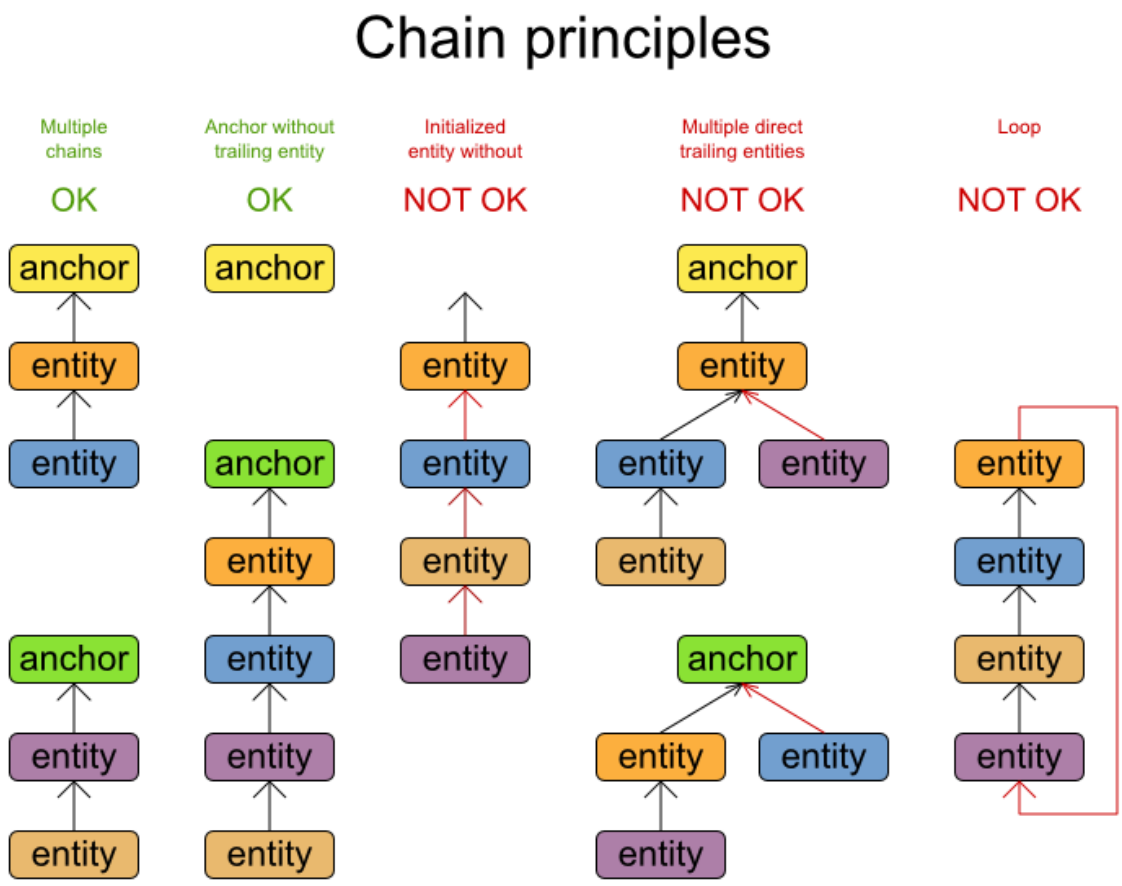
4.3.5.4. 連鎖型プランニング変数 (TSP、VRP など)

TSP や配送経路のユースケースには、連鎖が必要です。つまり、プランニングエンティティー同士で相互参照し、連鎖を形成します。(ツリー/ループのセットではなく) 連鎖セットとして問題をモデル化すると、探索空間を大幅に縮小することができます。

連鎖するプランニング変数は、以下のいずれかです。

- アンカー と呼ばれる問題ファクト (またはプランニングエンティティー) を直接参照する
- 同じプランニング変数を使用する別のプランニングエンティティーを参照して、再帰的にアンカーを参照する

以下に有効な連鎖と無効な連鎖の例を示します。



初期化されるプランニングエンティティーはすべて、アンカーから始まる、一端が開放型の連鎖に含まれます。有効なモデルとは以下のとおりです。

- 連鎖はループにはなりません。一端が必ず開いています。
- 連鎖にはすべてアンカーが1つだけ含まれます。このアンカーが問題ファクトで、プランニングエンティティーにはなりません。
- 連鎖はツリーにはならず、必ず1本の線となります。アンカーまたはプランニングエンティティーに続くプランニングエンティティーは、多くても1つだけです。
- 初期化されるプランニングエンティティーはすべて、連鎖に含まれます。
- プランニングエンティティーに参照されていないアンカーも連鎖とみなされます。

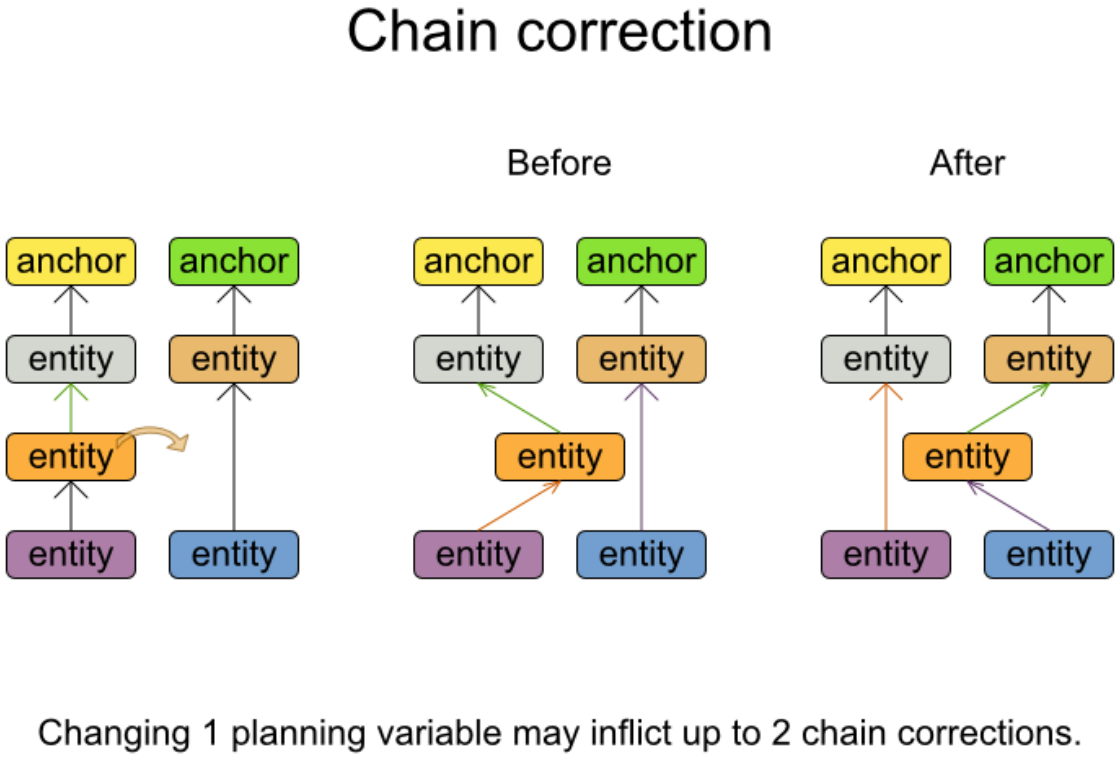
警告

Solver に渡される計画問題のインスタンスは、有効でなければなりません。

注記

制約が閉鎖型の連鎖を示す場合には、一端が開放された連鎖 (データベースで永続しやすい) としてモデル化し、最後のエンティティーをアンカーにつなげるスコア制約を実装します。

最適化アルゴリズムと組み込みの **Move** は、連鎖的に修正して、モデルが有効な状態を保てるようにします。



警告

カスタムの **Move** 実装は、モデルを有効な状態にする必要があります。

たとえば、TSP では、アンカーは **Domicile** です (配送経路ではアンカーは **Vehicle** です)。

```
public class Domicile ... implements Standstill {

    ...

    public City getCity() {...}

}
```

アンカー (つまり、問題ファクト) とプランニングエンティティーは、たとえば TSP の **Standstill** のように共通のインターフェースを実装します。

```
public interface Standstill {

    City getCity();

}
```

このインターフェースがプランニング変数の戻り値型で、さらにプランニング変数は連鎖されています (例: TSP の **Visit**、配送経路の **Customer**)。

```
@PlanningEntity
public class Visit ... implements Standstill {

    ...

    public City getCity() {...}

    @PlanningVariable(graphType = PlanningVariableGraphType.CHAINED,
        valueRangeProviderRefs = {"domicileRange", "visitRange"})
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }

    public void setPreviousStandstill(Standstill previousStandstill) {
        this.previousStandstill = previousStandstill;
    }

}
```

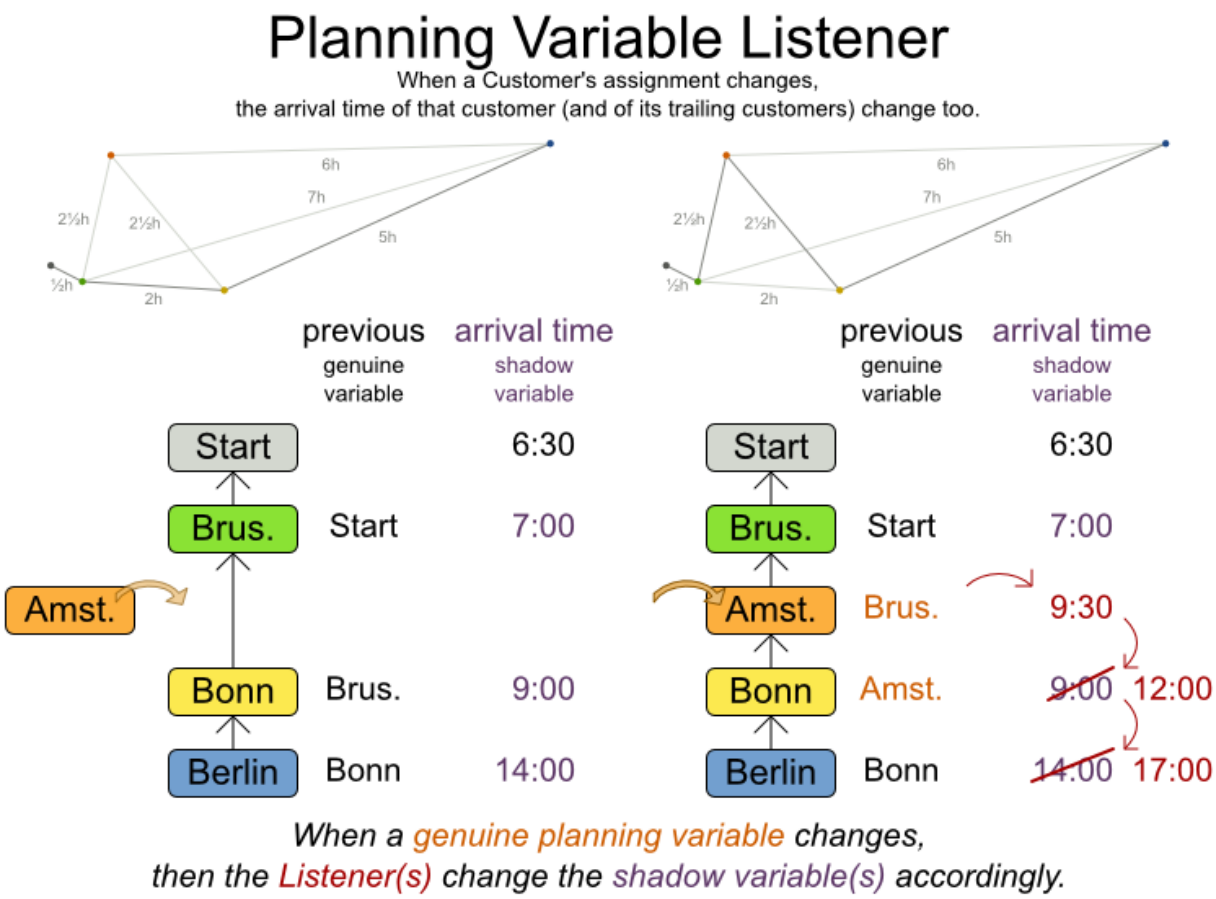
この 2 つの ValueRangeProvider が、通常、統合されている点に注目してください。

- **domicileList** など、アンカーを保持する ValueRangeProivder。
- **visitList** など、初期化されるプランニングエンティティーを保持する ValueRangeProvider。

4.3.6. シャドウ変数

4.3.6.1. 概要

シャドウ変数とは、正しい値を正規のプランニング変数の状態から推測できる変数のことです。この変数が定義上、正規化の原則に違反するにもかかわらず、ユースケースによっては、特により自然に制約を表現する場合など、シャドウ変数を使用すると非常に実用的です。たとえば、時間枠がある配送経路の場合には、車両が顧客先に到着する時間は、その車両がその前に顧客先に訪問したタイミング (および、2 箇所の既知の移動時間) をベースに計算することができます。



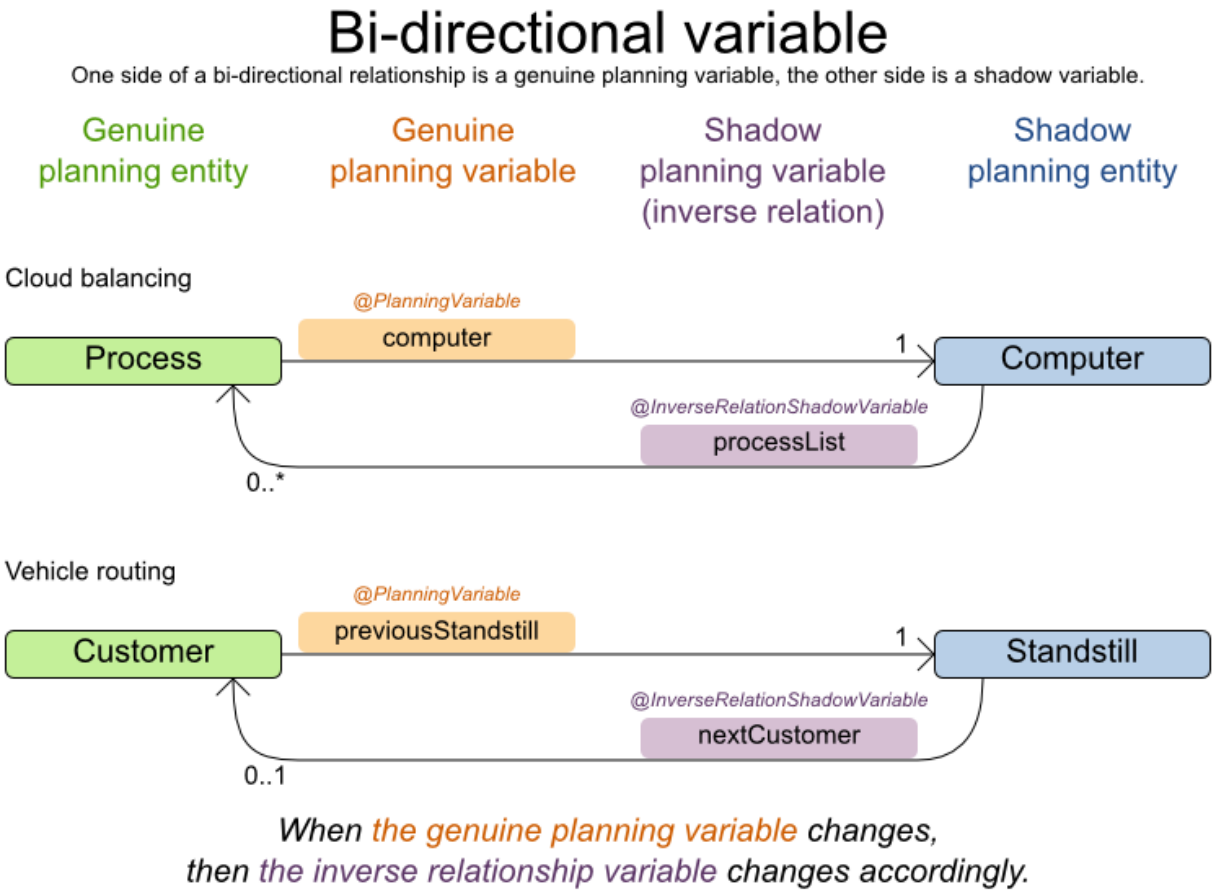
車両が対応する顧客が変化すると、各顧客先への到着時間は自動的に調節されます。詳しい情報は、「配送経路ドメインモデル」を参照してください。

スコア計算の観点からすると、シャドウ変数は他のプランニング変数とよく似ています。最適化の観点からすると、Planner は正規の変数を効果的に最適化するだけです (シャドウ変数はほぼ無視されます)。正規の変数が変化すると、従属のシャドウ変数もそれに合わせて変化するようにするだけです。

以下のように、組み込みのシャドウ変数は複数あります。

4.3.6.2. 双方向変数 (逆関係のシャドウ変数)

2 つの変数は、各変数のインスタンスが常に相互参照する場合には、双方向となります (一方が **null** を参照し、他方が存在しない場合を除きます)。そのため、A が B を参照している場合には、B は A を参照することになります。



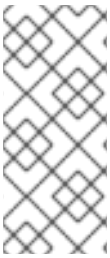
非連鎖型のプランニング変数の場合には、双方向の関係は、多対一の関係である必要があります。2 つのプランニング変数間で双方向の関係をマッピングするには、マスターの方 (正規の変数) を通常のプランニング変数としてアノテーションを付けます。

```
@PlanningEntity
public class CloudProcess {

    @PlanningVariable(...)
    public CloudComputer getComputer() {
        return computer;
    }
    public void setComputer(CloudComputer computer) {...}

}
```

次に、もう一方 (シャドウ変数) には、**Collection** (通常は **Set** または **List**) プロパティーで **@InverseRelationShadowVariable** アノテーションを付けます。

```
@PlanningEntity
public class CloudComputer {

    @InverseRelationShadowVariable(sourceVariableName = "computer")
    public List<CloudProcess> getProcessList() {
        return processList;
    }

}
```

sourceVariableName プロパティーは、ゲッターの戻り値型の正規プランニング変数の名前です (つまり、他方 の正規プランニング変数の名前)。



注記

シャドウプロパティー (**Collection**) は、**null** にすることはできません。そのシャドウエンティティーを参照する正規変数がない場合には、**Collection** は空になります。さらに、Solver が正規プランニング変数を初期化または変更しはじめると、それに合わせてそのシャドウ変数の **Collections** に対して追加/削除が行われるため、**Collection** は可変にする必要があります。

連鎖型のプランニング変数の場合には、双方向の関係は一対一にする必要があります。この場合、正規変数の側の設定は以下のようになります。

```
@PlanningEntity
public class Customer ... {

    @PlanningVariable(graphType = PlanningVariableGraphType.CHAINED, ...)
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }
    public void setPreviousStandstill(Standstill previousStandstill) {...}

}
```

また、シャドウ変数側の設定は以下のとおりです。

```
@PlanningEntity
public class Standstill {

    @InverseRelationShadowVariable(sourceVariableName = "previousStandstill")
    public Customer getNextCustomer() {
        return nextCustomer;
    }
    public void setNextCustomer(Customer nextCustomer) {...}

}
```

警告

Solver の入力計画問題は、双方向の関係を無視できません。A が B を参照する場合は、B が A を参照する必要があります。Planner は、計画時にこの原理に違反しませんが、入力もこの原理に違反することはできません。

4.3.6.3. アンカーシャドウ変数

アンカーシャドウ変数は、連鎖型変数 のアンカーのことです。

アンカーのプロパティーは、**@AnchorShadowVariable** でアノテーションを付けます。

```
@PlanningEntity
public class Customer {

    @AnchorShadowVariable(sourceVariableName = "previousStandstill")
    public Vehicle getVehicle() {...}
    public void setVehicle(Vehicle vehicle) {...}

}
```

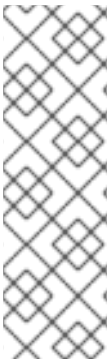
sourceVariableName プロパティーは、同じエンティティークラスの連鎖型変数の名前を指定します。

4.3.6.4. カスタムの VariableListener

Planner は、シャドウ変数を更新するのに **VariableListener** を使用します。また、カスタムのシャドウ変数を定義するには、カスタムの **VariableListener** を記述します。インターフェースを実装して、変更が必要なシャドウ変数にアノテーションを付けます。

```
    @PlanningVariable(...)
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }

    @CustomShadowVariable(variableListenerClass = VehicleUpdatingVariableListener.class,
            sources = {@CustomShadowVariable.Source(variableName = "previousStandstill")})
    public Vehicle getVehicle() {
        return vehicle;
    }
```

variableName は、シャドウ変数で変更をトリガーする変数を指定します。



注記

トリガー変数のクラスがシャドウ変数と異なる場合には、**@CustomShadowVariable.Source** で **entityClass** を指定します。この場合には、対象の **entityClass** が Solver 設定でプランニングエンティティークラスとして正しく設定されていることも確認します。正しく設定されていないと **VariableListener** はトリガーされません。

正規プランニング変数が含まれていなくても、シャドウ変数が最低 1 つ含まれるクラスは、プランニングエンティティークラスとなります。

たとえば **VehicleUpdatingVariableListener** は連鎖のアンカーの役割を果たし、連鎖内のすべての **Customer** に同じ **Vehicle** が指定されるようにします。

```
public class VehicleUpdatingVariableListener implements VariableListener<Customer> {

    public void afterEntityAdded(ScoreDirector scoreDirector, Customer customer) {
        updateVehicle(scoreDirector, customer);
    }

    public void afterVariableChanged(ScoreDirector scoreDirector, Customer customer) {
        updateVehicle(scoreDirector, customer);
    }
```

```
    ...

    protected void updateVehicle(ScoreDirector scoreDirector, Customer sourceCustomer) {
        Standstill previousStandstill = sourceCustomer.getPreviousStandstill();
        Vehicle vehicle = previousStandstill == null ? null : previousStandstill.getVehicle();
        Customer shadowCustomer = sourceCustomer;
        while (shadowCustomer != null && shadowCustomer.getVehicle() != vehicle) {
            scoreDirector.beforeVariableChanged(shadowCustomer, "vehicle");
            shadowCustomer.setVehicle(vehicle);
            scoreDirector.afterVariableChanged(shadowCustomer, "vehicle");
            shadowCustomer = shadowCustomer.getNextCustomer();
        }
    }

}
```

警告

VariableListener は、シャドウ変数のみを変更できます。正規プランニング変数や問題ファクトを変更することはできません。

警告

シャドウ変数を変更した場合は、**ScoreDirector** に通知する必要があります。

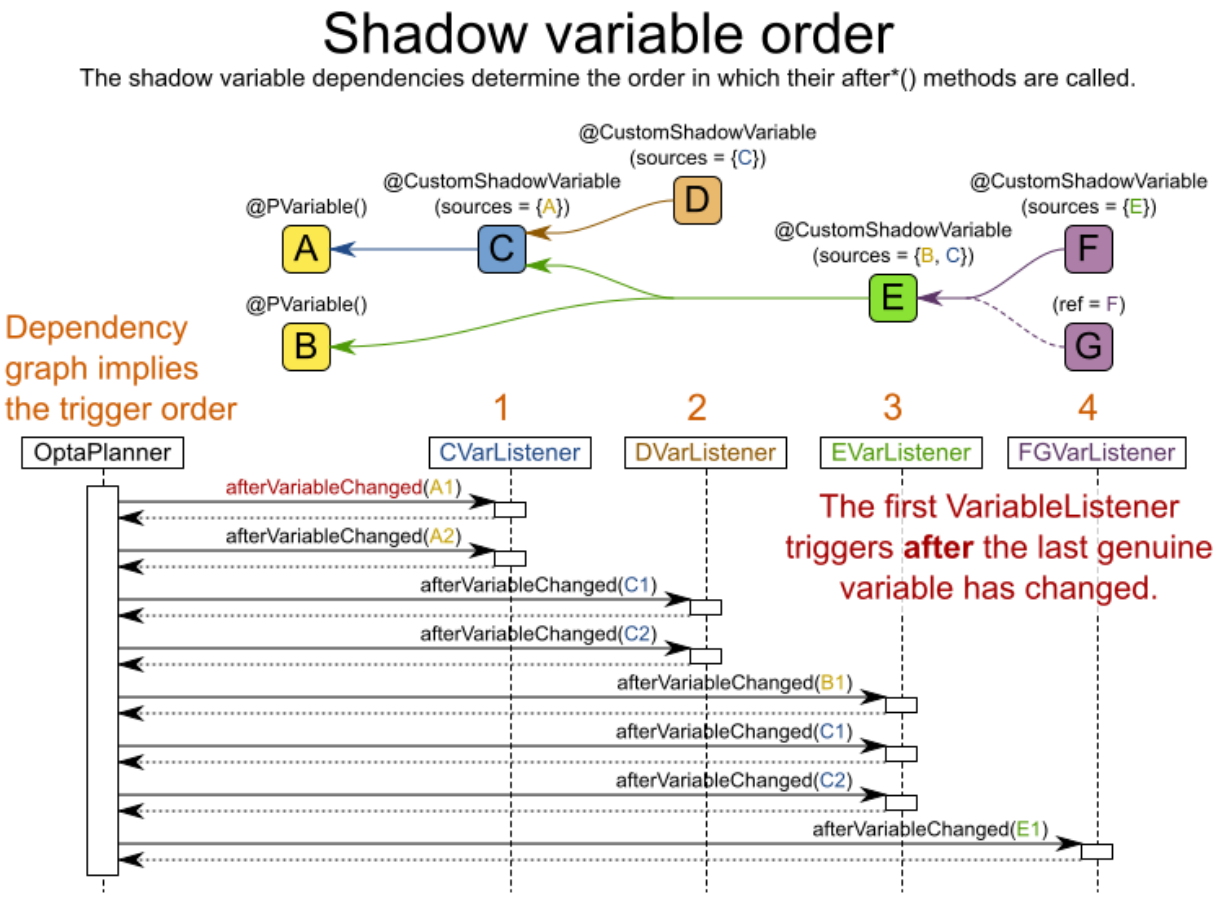
(**VariableListeners** を 2 つ設定するのは効率的でないため) 1 つの **VariableListener** が 2 つのシャドウ変数を変更する場合に、**variableListenerClass** でアノテーションを付けるのを 1 番目のシャドウ変数に限定し、他のシャドウ変数が 1 番目のシャドウ変数を参照するように設定します。

```
    @PlanningVariable(...)
    public Standstill getPreviousStandstill() {
        return previousStandstill;
    }

    @CustomShadowVariable(variableListenerClass =
TransportTimeAndCapacityUpdatingVariableListener.class,
            sources = {@CustomShadowVariable.Source(variableName = "previousStandstill")})
    public Integer getTransportTime() {
        return transportTime;
    }

    @CustomShadowVariable(variableListenerRef = @PlanningVariableReference(variableName =
"transportTime"))
```

```
public Integer getCapacity() {
    return capacity;
}
```

4.3.6.5. VariableListener による順番のトリガー

すべてのシャドウ変数は、組み込みまたはカスタムにかかわらず、**VariableListener** によりトリガーされます。正規の変数およびシャドウ変数でグラフが形成され、**afterEntityAdded()** メソッド、**afterVariableChanged()** メソッド、**afterEntityRemoved()** メソッドが呼び出される順番を決定します。



注記

上記の例では、D は E (または F) との間に直接的/間接的な依存関係がないので、E (または F) の後に指定することも可能です。

Planner は以下を保証します。

- 最初の **VariableListener after*()** メソッドは、最後の正規変数が変化した後にトリガーされます。正規変数 (上記の例では A と B) は、すべての **Move** が適用されているため、全インスタンス (上記の例では A1、A2、B1) で確実に整合性が取れた状態となっています。
- 2 番目の **VariableListener after*()** メソッドは、最初のシャドウ変数が変化した後にトリガーされます。最初のシャドウ変数 (上記の例では C) と正規変数も、全インスタンス (上記の例では値 C1、C2) で確実に整合性が取れた状態になっています。
- その他

影響を受けた順番になる可能性は高くなりますが、Planner は、異なるパラメーター (上記の例では A1 と A2) が指定した 同じ **VariableListener** に対して **after*()** メソッドが呼び出される順番を保証しません。

4.3.7. 計画問題およびその解

4.3.7.1. 計画問題インスタンス

計画問題のデータセットは、**Solver** が解 (ソリューション) を出せるように、クラス内でラッピングする必要があります。このクラスは必ず実装してください。たとえば N クィーン問題では、**Column** リスト、**Row** リスト、**Queen** リストが含まれる **NQueens** クラスです。

計画問題とは、未解決のプランニングソリューション、または 未初期化の ソリューション のことです。そのため、ラッピングクラスは **Solution** インターフェースを実装する必要があります。たとえば、N クィーン問題で、**NQueens** クラスは **Solution** を実装しますが、新しい **NQueens** に含まれる全 **Queen** は **Row** に割り当てられていません (クィーンの **row** プロパティーは **null** です)。これは実行可能解でも、可能解ではなく、未初期化解となります。

4.3.7.2. ソリューションインターフェース

Solver に対する **Solution** インスタンスとして問題を提示する必要があるため、クラスは **Solution** インターフェースを実装する必要があります。

```
public interface Solution<S extends Score> {

    S getScore();
    void setScore(S score);

    Collection<? extends Object> getProblemFacts();

}
```

たとえば、**NQueens** インスタンスは、全行、全列、全 **Queen** インスタンスの一覧を保持します。

```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {

    private int n;

    // Problem facts
    private List<Column> columnList;
    private List<Row> rowList;

    // Planning entities
    private List<Queen> queenList;

    // ...
}
```

また、プランニングソリューションクラスは、**@PlanningSolution** アノテーションを付ける必要があります。自動スキャン がない場合には、Solver 設定はプランニングソリューションクラスも宣言する必要があります。

```
<solver>
```

```
  ...
  <solutionClass>org.optaplanner.examples.nqueens.domain.NQueens</solutionClass>
  ...
</solver>
```

4.3.7.3. ソリューションからのエントリーの抽出

Planner は、**Solution** インスタンスからエンティティーインスタンスを抽出する必要があります。これは、**@PlanningEntityCollectionProperty** アノテーションが付いた全ゲッター (フィールド) を呼び出してこれらのコレクションを取得します。

```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {
    ...

    private List<Queen> queenList;

    @PlanningEntityCollectionProperty
    public List<Queen> getQueenList() {
        return queenList;
    }

}
```

@PlanningEntityCollectionProperty のアノテーションがついたメンバーが複数存在する可能性があります。このようなメンバーは、同じエンティティークラスタイプの **Collection** を返すことも可能です。

注記

@PlanningEntityCollectionProperty アノテーションは、@PlanningSolution アノテーションの付いたクラスに所属する必要があります。このアノテーションがないと、親クラスまたはサブクラスで無視されます。

あまり一般的ではありませんが、ゲッター (またはフィールド) の **@PlanningEntityProperty** を使用するなど、プランニングエンティティーがシングルトンの可能性があります。

4.3.7.4. getScore() および setScore() メソッド

Solution には、スコアプロパティーが必要です。スコアプロパティーは、**Solution** が初期化されていない場合や、スコアが (再) 計算されていない場合には **null** になります。**score** プロパティーは通常、使用する固有の **Score** 実装に型指定されます。たとえば、**NQueens** は **SimpleScore** を使用します。

```
@PlanningSolution
public class NQueens implements Solution<SimpleScore> {

    private SimpleScore score;

    public SimpleScore getScore() {
        return score;
    }

    public void setScore(SimpleScore score) {
        this.score = score;
```

```
    }

    // ...
}
```

代わりに、多くのユースケースでは、**HardSoftScore** を使用します。

```
@PlanningSolution
public class CourseSchedule implements Solution<HardSoftScore> {

    private HardSoftScore score;

    public HardSoftScore getScore() {
        return score;
    }

    public void setScore(HardSoftScore score) {
        this.score = score;
    }

    // ...
}
```

Score 実装に関する詳しい情報は、スコア計算のセクションを参照してください。

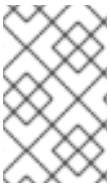
4.3.7.5. getProblemFacts() メソッド

このメソッドは、スコア計算に Drools を使用する場合にのみ使用します。他のスコアディレクターは、このメソッドを使用しません。

getProblemFacts() メソッドが返すオブジェクトはすべて、Drools のワーキングメモリーにアサートされるので、スコアルールがそのオブジェクトにアクセスできます。たとえば、**NQueens** は **Column** インスタンスと **Row** インスタンスだけを返します。

```
    public Collection<? extends Object> getProblemFacts() {
        List<Object> facts = new ArrayList<Object>();
        facts.addAll(columnList);
        facts.addAll(rowList);
        // Do not add the planning entities (queenList) because that will be done automatically
        return facts;
    }
```

プランニングエンティティーはすべて自動的に Drools のワーキングメモリーに挿入されます。**getProblemFacts()** メソッドには追加しないでください。



注記

よくある間違いとして、**Collection** に対して **fact.addAll(…)** ではなく、**facts.add(…)** を使用し、**Collection** の要素が Drools のワーキングメモリーに含まれていないため、スコアルールと照合できなくなってしまいます。

getProblemFacts() メソッドは、Solver スレッドの各 Solver フェーズで呼び出されるのは 1 度だけで、頻繁には呼び出されません。

4.3.7.5.1. キャッシュされた問題ファクト

キャッシュされた問題ファクトとは、実際のドメインモデルには存在せずに、**Solver** が実際に問題を解決し始める前に計算される問題ファクトのことです。**getProblemFacts()** メソッドは、キャッシュされた問題ファクトを使用してドメインモデルを改良できるため、スコア制約の簡素化、迅速化につながる可能性があります。

たとえば、キャッシュされた問題ファクト **TopicConflict** は、**Student** を少なくても 1 つ共有する 2 つの **Topics** に対して作成されます。

```
public Collection<? extends Object> getProblemFacts() {
    List<Object> facts = new ArrayList<Object>();
    // ...
    facts.addAll(calculateTopicConflictList());
    // ...
    return facts;
}

private List<TopicConflict> calculateTopicConflictList() {
    List<TopicConflict> topicConflictList = new ArrayList<TopicConflict>();
    for (Topic leftTopic : topicList) {
        for (Topic rightTopic : topicList) {
            if (leftTopic.getId() < rightTopic.getId()) {
                int studentSize = 0;
                for (Student student : leftTopic.getStudentList()) {
                    if (rightTopic.getStudentList().contains(student)) {
                        studentSize++;
                    }
                }
                if (studentSize > 0) {
                    topicConflictList.add(new TopicConflict(leftTopic, rightTopic, studentSize));
                }
            }
        }
    }
    return topicConflictList;
}
```

スコア制約を使用して、学生を共有するトピックの試験が、近い時間帯でスケジュールされないように確認する必要がある場合には (同時、続けて、または同じ日など制約によって異なります)、2 つの **Student** インスタンスに統合するのではなく、**TopicConflict** インスタンスを問題ファクトとして使用できます。

4.3.7.6. ソリューションのクローン作成

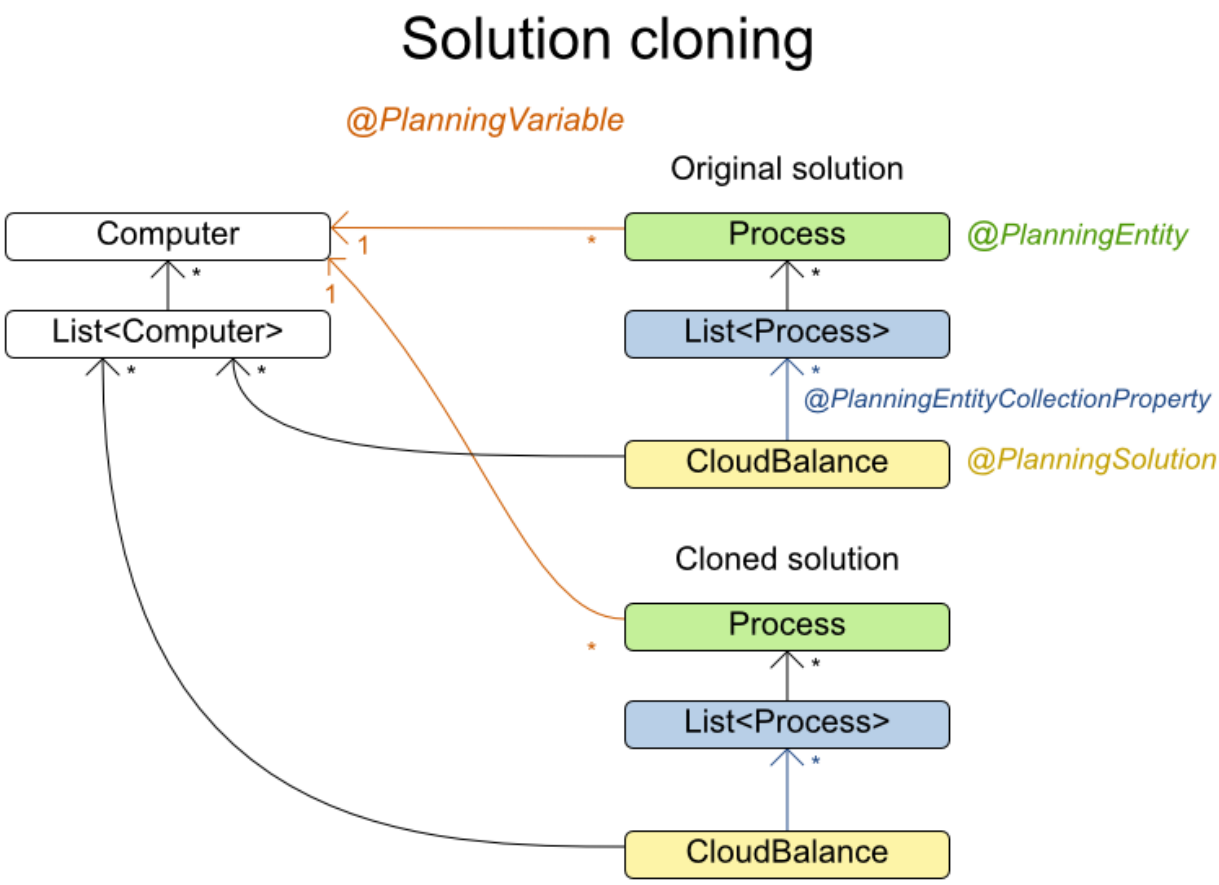
(すべてではないとしても) 多くの最適化アルゴリズムは、新しい最適解を見つけた場合 (後で呼び出すため)、または複数の解を並行して使用できるように、その解 (ソリューション) のクローンを作成します。

注記

シャロークローン、ディープクローンなど、クローンには多数の方法があります。このコンテキストでは、計画クローン にフォーカスします。

ソリューション の計画クローンは、以下の要件を満たす必要があります。

- クローンは同じ計画問題を表現する必要があります。通常、元のものと同じ問題ファクトおよび問題ファクトコレクションのインスタンスを再利用します。
- クローンは、エンティティーおよびエンティティーコレクションをクローンした異なるインスタンスを使用する必要があります。元の ソリューション エンティティーの変数に変更を加えても、クローンに影響があってはいけません。



計画クローンメソッドを実装するのは困難なので、実装する必要はありません。

4.3.7.6.1. FieldAccessingSolutionCloner

この **SolutionCloner** はデフォルトで使用されます。多くのユースケースと適切に機能します。

警告

FieldAccessingSolutionCloner がエンティティーコレクションのクローンを作成する場合には、実装が認識されずに **ArrayList**、**LinkedHashSet** または **TreeSet** (より適切なもの) に置き換えられる可能性があります。ただし、一般的な JDK **Collection** 実装の大半は認識されます。

FieldAccessingSolutionCloner は、デフォルトでは問題ファクトのクローンを作成しません。問題ファクトにプランニングエンティティーまたはプランニングソリューションを参照させるなど、問題

ファクトをデープクローンして計画クローンにする必要がある場合には、以下のように **@DeepPlanningClone** アノテーションを付けます。

```
@DeepPlanningClone
public class SeatDesignationDependency {
    private SeatDesignation leftSeatDesignation; // planning entity
    private SeatDesignation rightSeatDesignation; // planning entity
    ...
}
```

上記の例のように、**SeatDesignation** はプランニングエンティティー (つまり、自動的にディープ計画クローンされる) ので、**SeatDesignationDependency** もディープ計画クローンする必要があります。

または **@DeepPlanningClone** アノテーションをゲッターメソッドで使用することも可能です。

4.3.7.6.2. カスタムクローン: ソリューションでの PlanningCloneable の実装

Solution が **PlanningCloneable** を実装する場合に、**planningClone()** メソッドを呼び出すと自動的に Planner がクローンします。

```
public interface PlanningCloneable<T> {

    T planningClone();

}
```

例: **NQueens** が **PlanningCloneable** を実装する場合には、すべての **Queen** インスタンスをディープクローンするだけです。計画中に クィーン を変更して元のソリューションが変更された場合には、このクローンは変化しません。

```
public class NQueens implements Solution<...>, PlanningCloneable<NQueens> {
    ...

    /**
     * Clone will only deep copy the {@link #queenList}.
     */
    public NQueens planningClone() {
        NQueens clone = new NQueens();
        clone.id = id;
        clone.n = n;
        clone.columnList = columnList;
        clone.rowList = rowList;
        List<Queen> clonedQueenList = new ArrayList<Queen>(queenList.size());
        for (Queen queen : queenList) {
            clonedQueenList.add(queen.planningClone());
        }
        clone.queenList = clonedQueenList;
        clone.score = score;
        return clone;
    }
}
```

planningClone() メソッドは、プランニングエンティティーだけをディープクローンする必要があります。**Column** および **Row** などの問題ファクトは通常クローンされない点に注意してください。さらに、これらの **List** インスタンスもクローンされません。問題ファクトもクローンする場合には、新し

いプランニングエンティティークローンが、ソリューションにより使用される新しい問題ファクトのクローンを参照しているようにする必要があります。たとえば、すべての **Row** インスタンスのクローンを作成する場合には、各 **Queen** および **NQueens** のクローン自体は、これらの新しい **Row** クローンを参照する必要があります。

警告

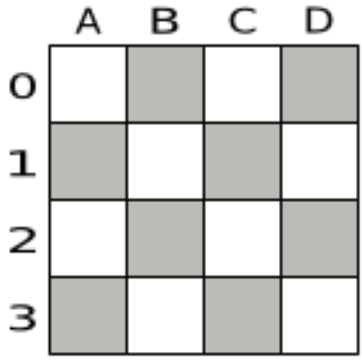
連鎖型 変数でエンティティーをクローンする方法は正しくありません。エンティティー A の変数は別のエンティティー B に参照します。A がクローンされる場合には、A の変数は、元の B ではなく、B のクローンを参照する必要があります。

4.3.7.7. 初期化されていないソリューションの作成

Solution インスタンスを作成して、計画問題のデータセットを表現するため、解決する計画問題として、**Solver** に設定することができます。たとえば N クィーンでは、**NQueens** インスタンスが、必須の **Column** インスタンスおよび **Row** インスタンスで作成され、全 **Queen** は異なる **column** に、全 **row** は **null** に設定されます。

```
private NQueens createNQueens(int n) {
    NQueens nQueens = new NQueens();
    nQueens.setId(0L);
    nQueens.setN(n);
    nQueens.setColumnList(createColumnList(nQueens));
    nQueens.setRowList(createRowList(nQueens));
    nQueens.setQueenList(createQueenList(nQueens));
    return nQueens;
}

private List<Queen> createQueenList(NQueens nQueens) {
    int n = nQueens.getN();
    List<Queen> queenList = new ArrayList<Queen>(n);
    long id = 0L;
    for (Column column : nQueens.getColumnList()) {
        Queen queen = new Queen();
        queen.setId(id);
        id++;
        queen.setColumn(column);
        // Notice that we leave the PlanningVariable properties on null
        queenList.add(queen);
    }
    return queenList;
}
```

図4.3 クィーン 4 個のパズルの初期化前の解



通常、データ層から来るこのデータや **Solution** 実装の多くは、対象のデータを累積して、計画する未初期化プランニングエンティティーインスタンスを作成します。

```
    private void createLectureList(CourseSchedule schedule) {
        List<Course> courseList = schedule.getCourseList();
        List<Lecture> lectureList = new ArrayList<Lecture>(courseList.size());
        long id = 0L;
        for (Course course : courseList) {
            for (int i = 0; i < course.getLectureSize(); i++) {
                Lecture lecture = new Lecture();
                lecture.setId(id);
                id++;
                lecture.setCourse(course);
                lecture.setLectureIndexInCourse(i);
                // Notice that we leave the PlanningVariable properties (period and room) on null
                lectureList.add(lecture);
            }
        }
        schedule.setLectureList(lectureList);
    }
```

4.4. SOLVER の使用

4.4.1. Solver インターフェース

Solver を実装し、計画問題を解決します。

```
public interface Solver<S extends Solution> {

    S solve(S planningProblem);

    ...
}
```

Solver が一度に解決できるのは、1 つの計画問題インスタンスだけです。**Solver** は、1 つのスレッドからのみアクセスするようにしてください。ただし、特にスレッドセーフとして Javadoc に記述されているメソッドは除きます。**SolverFactory** で構築しているため、自分で実装する必要はありません。

4.4.2. 問題の解決

以下があれば、問題の解決は非常に簡単です。

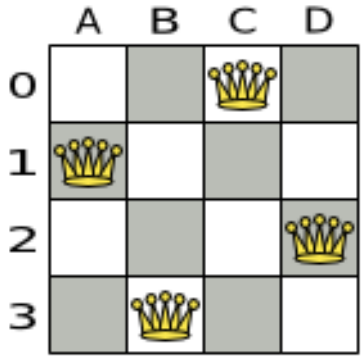
- Solver 設定からビルドした **Solver**
- 計画問題インスタンスを示す ソリューション

計画問題を、**solve()** メソッドへの引数として指定すると、見つかった最適解が返されます。

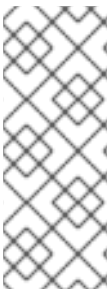
```
NQueens bestSolution = solver.solve(planningProblem);
```

たとえば N クィーンでは、**solve()** メソッドは、**Row** に割り当てられたすべての クィーン を含む **NQueens** インスタンスを返します。

図4.4 8ms でクィーン 4 個のパズルに対する最善解 (および最適解)



solve(Solution) メソッドは、(問題のサイズや Solver 設定によって) 時間がかかる場合があります。**Solver** は、可能解の 探索空間 からインテリジェントに情報を取得し、問題解決時に見つけた最善解を記憶します。さまざまな要因によっては (問題サイズ、**Solver** の所有時間 (許容範囲)、Solver 設定など)、最善解が、最適解となる場合と、ならない場合があります。



注記

solve(Solution) メソッドに渡される **Solution** インスタンスは、**Solver** によって変更しますが、最善解ではないので間違わないようにしてください。

solve(Solution) または **getBestSolution()** が返す **Solution** インスタンスは、**solve(Solution)** メソッドに渡すインスタンスの 計画クローン である可能性が最も高くなります。つまり、これは別のソリューションということになります。

注記

solve(Solution) メソッドに渡す **Solution** インスタンスは、初期化する必要はありません。一部または完全に初期化することは可能で、反復計画 の場合に頻繁に該当します。

4.4.3. 環境モード: コードにおける問題の有無

環境モードでは、実装内の共通のバグを検出できます。このモードによるロギングレベルに影響はありません。

以下のように、Solver 設定の XML ファイルで環境モードを設定します。

```
<solver>
  <environmentMode>FAST_ASSERT</environmentMode>
```

```
  ...
</solver>
```

Solver には **Random** インスタンスが 1 つ含まれます。Solver の設定によっては、他よりもはるかに多い **Random** インスタンスを使用するものもあります。たとえば、焼きなまし法は、乱数に大きく依存していますが、タブー探索はスコアとの紐付けを処理する場合にのみ依存します。環境モードは、**Random** インスタンスのシードに影響を与えます。

以下は環境モードです。

4.4.3.1. FULL_ASSERT

FULL_ASSERT モードは (インクリメンタルスコアの計算を Move ごとに低下させないようにするアサーションなど) すべてのアサーションをオンにして、Move 実装、スコアルール、ルールエンジン自体などでバグがあった場合にフェイルファストさせます。

このモードは、再現可能です (再現可能モードを参照)。また、アサーション以外のモードと比べ、**calculateScore()** メソッドは頻繁に呼び出されるため、割り込みも可能です。

FULL_ASSERT モードは、(インクリメンタルスコアの計算には依存しないので) 極めて遅いです。

4.4.3.2. NON_INTRUSIVE_FULL_ASSERT

NON_INTRUSIVE_FULL_ASSERT は、複数のアサーションをオンにして、Move 実装、スコアルール、ルールエンジン自体などにバグがあった場合にフェイルファストさせます。

このモードは、再現可能です (再現可能モードを参照)。また、アサーション以外のモードと比べ、**calculateScore()** メソッドはより頻繁に呼び出されないので、割り込みはしません。

NON_INTRUSIVE_FULL_ASSERT モードは、(インクリメンタルスコアの計算には依存しないので) 極めて遅いです。

4.4.3.3. FAST_ASSERT

FAST_ASSERT モードは、(undoMove のスコアが Move の前と同じであるとのアサーションなど) 大半のアサーションをオンにして、Move 実装、スコアルール、ルールエンジン自体などでバグがあった場合にフェイルファストさせます。

このモードは、再現可能です (再現可能モードを参照)。また、アサーション以外のモードと比べ、**calculateScore()** メソッドがより頻繁に呼び出されるので、割り込みが可能です。

FAST_ASSERT モードは遅いです。

FAST_ASSERT モードがオンの状態で計画問題を簡略的に実行するテストケースを記述することを推奨します。

4.4.3.4. REPRODUCIBLE (デフォルト)

再現可能モードは、開発中に推奨されるモードであるため、デフォルトのモードとなっています。このモードでは、Planner バージョンを実行した場合に実行されるコードの順番は常に同じです。そして、以下の注意点が適用されている場合を除き、どのステップでも同じ結果となります。こうすることで、一貫性を持ってバグを再現することができます。したがって、実行を繰り返しながら、特定のリファクタリング (例: スコア制約のパフォーマンスの最適化) をベンチマーク化することができます。

注記

再現可能モードでも、以下の理由から、アプリケーションを完全に再現できない可能性があります。

- 特に ソリューション の実装で、プランニングエンティティーまたは計画値 (ただし一般的な問題ファクトでないもの) のコレクションに **HashSet** (または、JVM 実行間の順番に一貫性のない別の コレクション) を使用しているため。これは、**LinkedHashSet** に置き換えます。
- 時間の勾配に依存するアルゴリズム (特に焼きなまし法) が Time Spent Termination (所要時間による終了) と組み合わせられているため。割り当てられた CPU 時間に大きな相違があると、時間勾配の値にも影響があります。焼きなまし法をレイトアクセプタンスに置き換えるか、Time Spent Termination (所要時間による終了) を Step Count Termination (ステップ数による終了) に置き換えます。

再現可能モードは、実稼働モードより若干遅くなります。実稼働環境で再現性が必要な場合には、このモードを実稼働環境でも使用してください。

シードを指定しない場合は、このモードではデフォルトで固定の 乱数シード が使用され、特定の同時最適化 (例: ワークスティーリング) も無効にします。

4.4.3.5. PRODUCTION

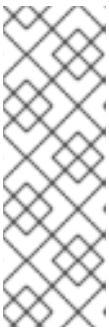
実稼働モードは、最速ですが再現性はありません。このモードは、再現性の必要がない、実稼働環境に推奨されます。

シードを指定しない場合は、このモードでは固定の 乱数シード は使用されません。

4.4.4. ログレベル: Solver の機能

Solver の理解を容易にする最適な方法として、ロギングレベルを使用することが挙げられます。

- error: **RuntimeException** として呼び出しコードに送出されるエラー以外のエラーをログに記録します。

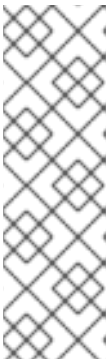


注記

エラーが発生した場合には、Planner は通常、呼び出しコードに対する詳細メッセージを含む **RuntimeException** のサブクラスを送出し、フェイルファスト (処理が中断) されます。ログメッセージが重複しないように、これはエラーとしてログに記録されません。呼び出しコードがこの **RuntimeException** を明示的に受けて処理した場合を除き、**Thread** のデフォルトの **ExceptionHandler** は、エラーとしてログに記録されます。一方で、被害が大きくなったり、エラーが複雑になり分かりにくくならないように、コードが中断されます。

- warn: 不審な状況をログに記録します。
- info: すべてのフェーズおよび Solver 自体をログに記録します。「スコープの概要」を参照してください。
- debug: 全フェーズの全ステップをログに記録します。「スコープの概要」を参照してください。

- **trace**: 全フェーズの全ステップに含まれる全 Move をログに記録します。「スコープの概要」を参照してください。



注記

trace ログをオンにすると、パフォーマンスが大幅に低下します。通常は、4 倍ほど遅くなります。ただし、開発中にボトルネックを発見するのに非常に有効です。

デバッグログでさえも、(レイトアクセプタンスや焼きなまし法などの) ステップの動きが速いアルゴリズムでは、パフォーマンスが大幅に低下する可能性がありますが、(タブー探索などの) ステップの動きが遅いアルゴリズムはパフォーマンスでは低下しません。

たとえば、**debug** ログに設定して、フェーズが終了するタイミングや、ステップの速度を確認してみます。

```
INFO  Solving started: time spent (3), best score (uninitialized/0), random (JDK with seed 0).
DEBUG     CH step (0), time spent (5), score (0), selected move count (1), picked move (Queen-2 {null -> Row-0}).
DEBUG     CH step (1), time spent (7), score (0), selected move count (3), picked move (Queen-1 {null -> Row-2}).
DEBUG     CH step (2), time spent (10), score (0), selected move count (4), picked move (Queen-3 {null -> Row-3}).
DEBUG     CH step (3), time spent (12), score (-1), selected move count (4), picked move (Queen-0 {null -> Row-1}).
INFO  Construction Heuristic phase (0) ended: step total (4), time spent (12), best score (-1).
DEBUG     LS step (0), time spent (19), score (-1),     best score (-1), accepted/selected move count (12/12), picked move (Queen-1 {Row-2 -> Row-3}).
DEBUG     LS step (1), time spent (24), score (0), new best score (0), accepted/selected move count (9/12), picked move (Queen-3 {Row-3 -> Row-2}).
INFO  Local Search phase (1) ended: step total (2), time spent (24), best score (0).
INFO  Solving ended: time spent (24), best score (0), average calculate count per second (1625).
```

かかった時間はすべてミリ秒単位で表示されています。

すべてのログが SLF4J に記録されます。SLF4J とは、シンプルなログファサードで、Logback、Apache Commons Logging、Log4j、または java.util.logging にすべてのログメッセージを委ねます。任意のログフレームワークのログアダプターに依存関係を追加します。

まだどのログフレームワークも使用していない場合には、以下の Maven の依存関係を追加して Logback を使用してください (別のブリッジ依存関係を追加する必要はありません)。

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.x</version>
</dependency>
```

logback.xml ファイルで **org.optaplanner** パッケージのログレベルを設定します。

```
<configuration>

  <logger name="org.optaplanner" level="debug"/>
```

```
  ...

<configuration>
```

Logback ではなく Log4J 1.x を使用している場合 (かつ、高速な後継バージョンである Logback に切り替えない場合) には、ブリッジ依存関係を追加します。

```
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.x</version>
  </dependency>
```

さらに、**log4j.xml** ファイルで **org.optaplanner** パッケージにログレベルを設定します。

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <category name="org.optaplanner">
    <priority value="debug" />
  </category>

  ...

</log4j:configuration>
```

注記

マルチテナント型のアプリケーションでは、複数の **Solver** インスタンスが同時に実行する可能性があります。ログを別のファイルに分けるには、MDC で **solve()** 呼び出しを囲んでください。

```
        MDC.put("tenant.name", tenantName);
        Solution bestSolution = solver.solve(planningProblem);
        MDC.remove("tenant.name");
```

次に **${tenant.name}** ごとに異なるファイルを使用するロガーを設定します。たとえば、Logback の場合は、**logback.xml** で **SiftingAppender** を使用します。

```
<appender name="fileAppender" class="ch.qos.logback.classic.sift.SiftingAppender">
  <discriminator>
    <key>tenant.name</key>
    <defaultValue>unknown</defaultValue>
  </discriminator>
  <sift>
    <appender name="fileAppender.${tenant.name}" class="...FileAppender">
      <file>local/log/optaplanner-${tenant.name}.log</file>
      ...
    </appender>
  </sift>
</appender>
```

4.4.5. 乱数生成器

多くのヒューリスティックやメタヒューリスティックは、Move の選択に擬似乱数生成器を使用し、スコアの連携、確立ベースの Move アクセプタンスなどを解決します。問題解決時、同じ **Random** インスタンスが再利用され、再現性、パフォーマンス、乱数値の均等分布を向上させます。

Random インスタンスの乱数シードを変更するには、以下のように **randomSeed** を指定します。

```
<solver>
  <randomSeed>0</randomSeed>
  ...
</solver>
```

疑似乱数生成器の実装を変更するには、以下のように **randomType** を指定します。

```
<solver>
  <randomType>MERSENNE_TWISTER</randomType>
  ...
</solver>
```

以下のタイプがサポートされます。

- **JDK** (デフォルト): 標準実装 (**java.util.Random**).
- **MERSENNE_TWISTER**: Commons Math による実装
- **WELL512A**、**WELL1024A**、**WELL19937A**、**WELL19937C**、**WELL44497A**、および **WELL44497B**: Commons Math による実装

多くのユースケースでは、複数のデータセットにおける最善解の平均的な品質に対して、randomType からの影響は全くありません。実際のユースケースでこの点を確認する場合は、ベンチマーカー をご利用ください。

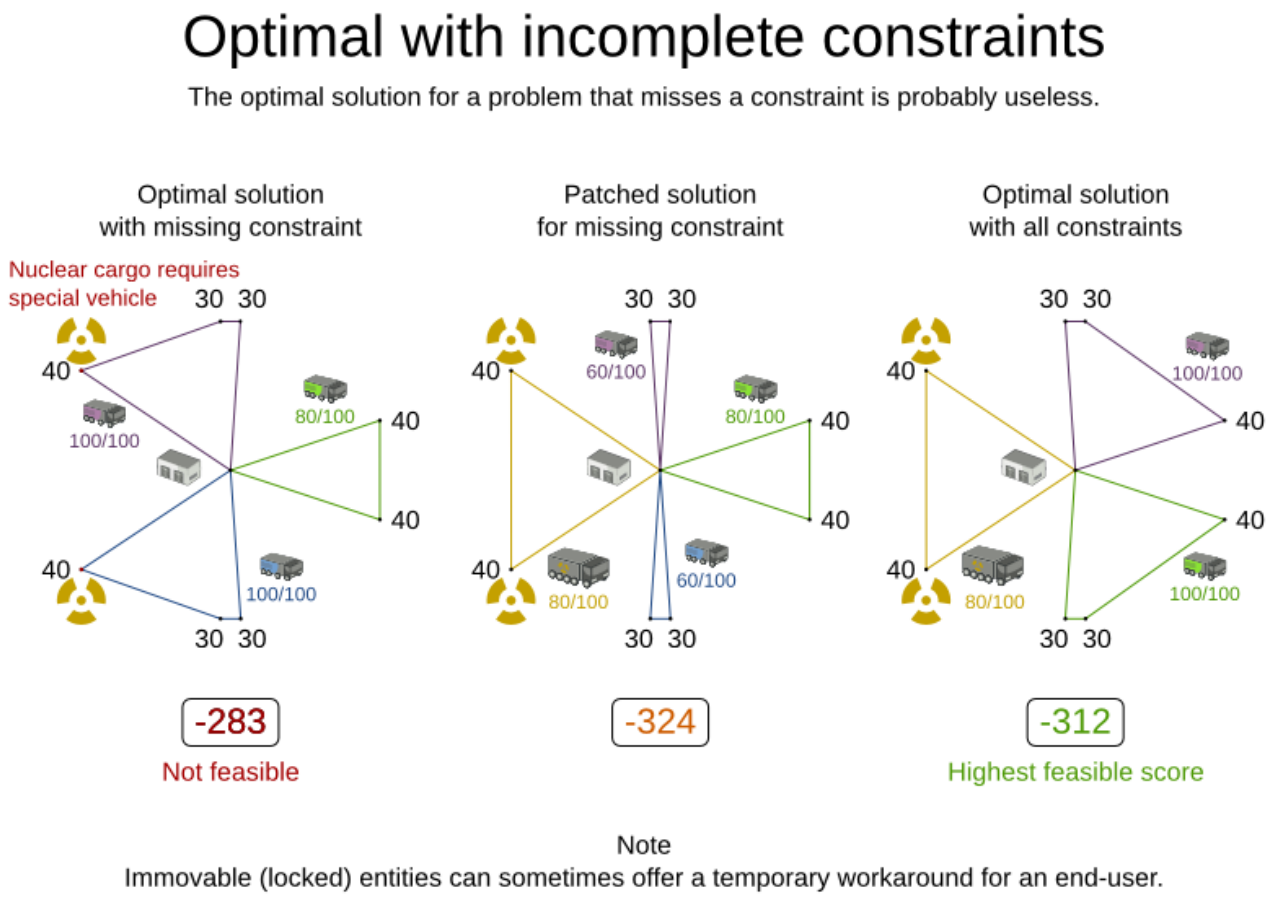
第5章 スコア計算

5.1. スコアに関する用語

5.1.1. スコアについて

初期化された ソリューション には常にスコアがあります。スコアは、2 つのソリューションを客観的に比較する方法の 1 つです。スコアの高いソリューションのほうが優れています。**Solver** は、考えられるソリューションの中でスコア が一番高い ソリューション を見つけることを目的とします。最善解は、問題解決時に **Solver** が特定した、スコア が一番高い ソリューション のことで、これが 最適解 となる場合もあります。

Planner は、どの ソリューション が特定のビジネスに適しているかを自動的に把握することはできないため、ビジネスのニーズに合わせて、特定の ソリューション のスコアを計算する方法を指定する必要があります。重要なビジネス制約を実装できない場合や、実装するのを忘れた場合、このソリューションはおそらく役に立ちません。

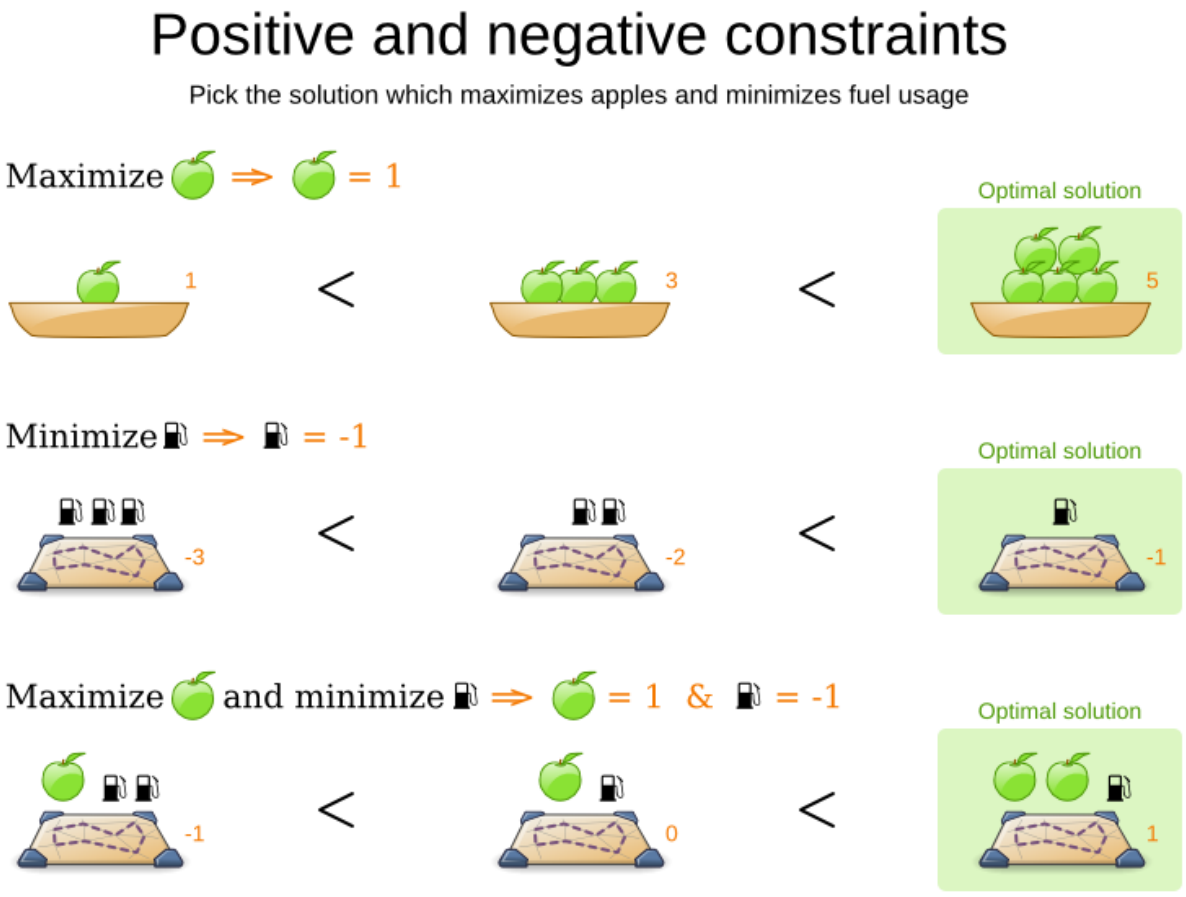


以下のスコア手法を使用すると、Planner での制約を非常に柔軟に定義できます。

- スコアの符号 (正または負): 制約型を最大化または最小化します。
- スコアの重みづけ: 制約型にコスト/利益を追加します。
- スコアレベル (ハード、ソフトなど): 制約型のグループに優先順位を付けます。
- パレートスコアリング

5.1.2. スコア制約の符号 (正または負)

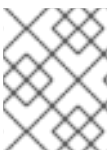
すべてのスコア手法は、制約をベースにしています。制約は (ソリューションの中でりんごの収穫を最大化するなど) 単純なパターンの場合も、より複雑なパターンの場合もあります。正の制約は、最大化させる制約のことで、負の制約は最小化させる制約を指します。



上の画像では、制約が正でも負でも、常にスコアが最高となる、最適解について例示しています。

計画問題の多くには、負の制約のみが含まれるので、スコアは負になります。このような場合には、完全スコアを **0** とし、違反した負の制約の重みを合計した値がスコアになります。たとえば、「4 個のクィーン」のソリューションも、相互に攻撃可能なクィーンペアの数を負にした値となります。

同じスコアレベルでも、負および正の制約を組み合わせることができます。



注記

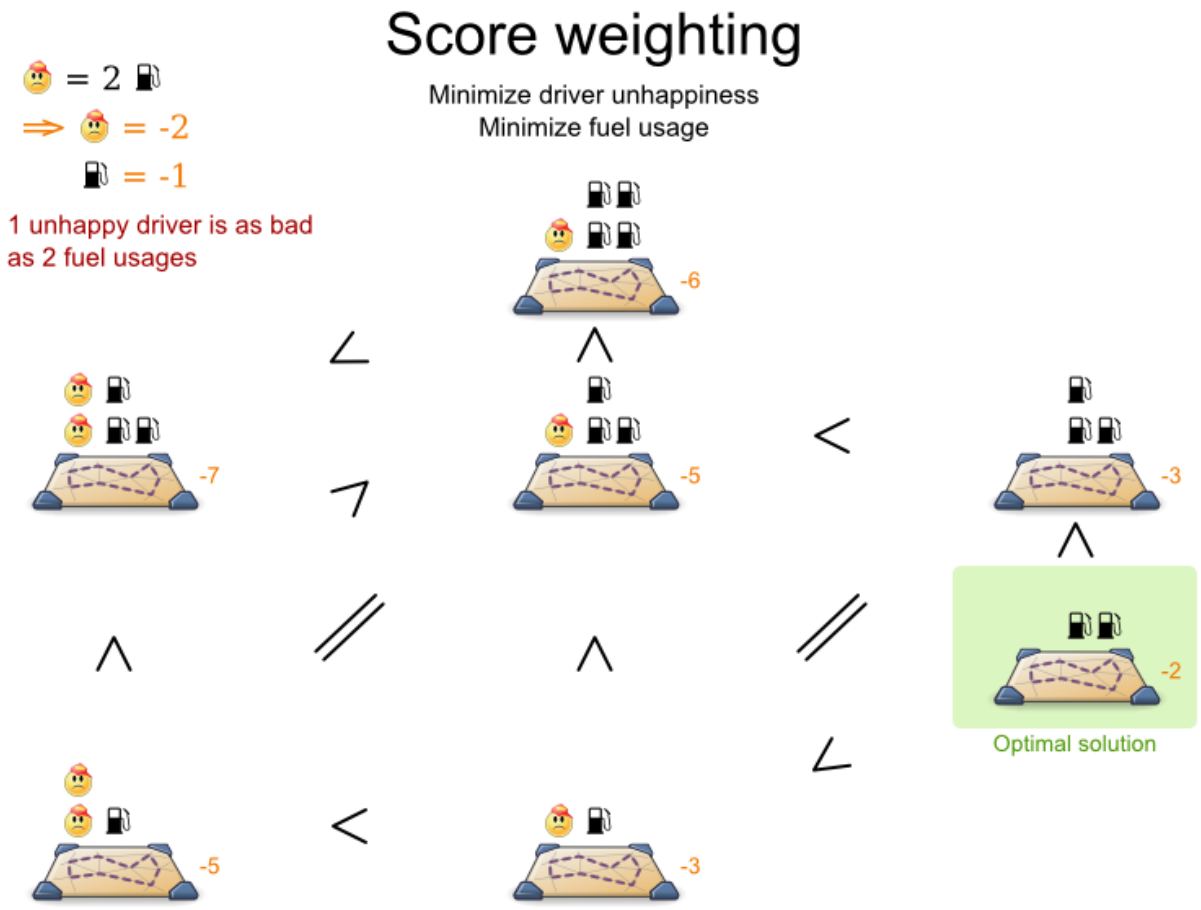
ビジネスがすべてのスコア制約を事前に把握していると思い込まないでください。最初のリリース以降でも、スコア制約が追加、変更されることを想定してください。

特定のプランニングエンティティーセットで、(負制約の違反や、正制約の遵守により) 制約が有効になった場合のことを 制約の一致 と呼びます。

5.1.3. スコア制約の重みづけ

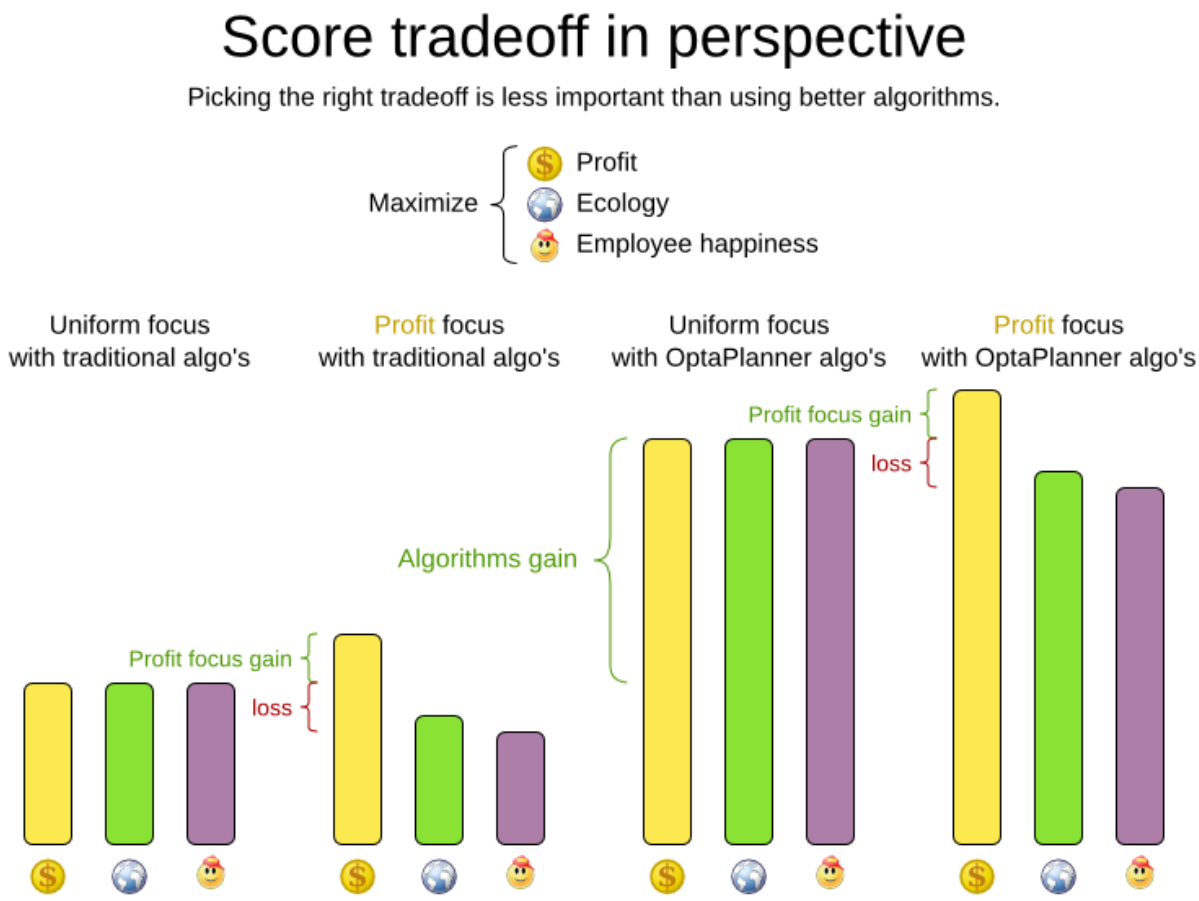
スコア制約の重要性は、すべて均等ではありません。ある制約 1 件に違反する場合と、別の制約を複数回違反する場合に、違反の程度が同じだとすれば、この 2 つの制約における重みは異なることになります (ただし、スコアレベルは同じです)。たとえば、配送経路問題では、「不満があるドライバー」制約

の一致数1件と、「ガソリンタンクの使用量」制約の一致数2件の重みが同じになるように設定できます。

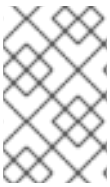


スコアの重みは、すべてに値段を付けられるユースケースで頻繁に使用されます。このような場合、正制約は収益を最大化し、負制約は経費を最小化するため、両方をあわせて利益を最大化します。スコアの重みづけは、社会的な 公平 を作り出す際にもよく使用されます。たとえば、「看護師」の例では、通常の日ではなく、大晦日に公休日を希望する場合、重みが大きくなります。

他の制約との関連を考慮して選択していくため、分析の面から考えると、制約に適切な重みを課すのは困難になる可能性があります。ただし、重みが正確ではないときの損害の大きさは、アルゴリズムが優れていないのに比べると少なくなります。



使用するプランニングエンティティーをベースに、制約一致の重みを動的に変更させることもできます。たとえば、「クラウドのバランス」の例でにおける、有効な コンピューター に対するソフト制約一致の重みは、その コンピューター の コスト になります (コンピューターごとに異なります)。



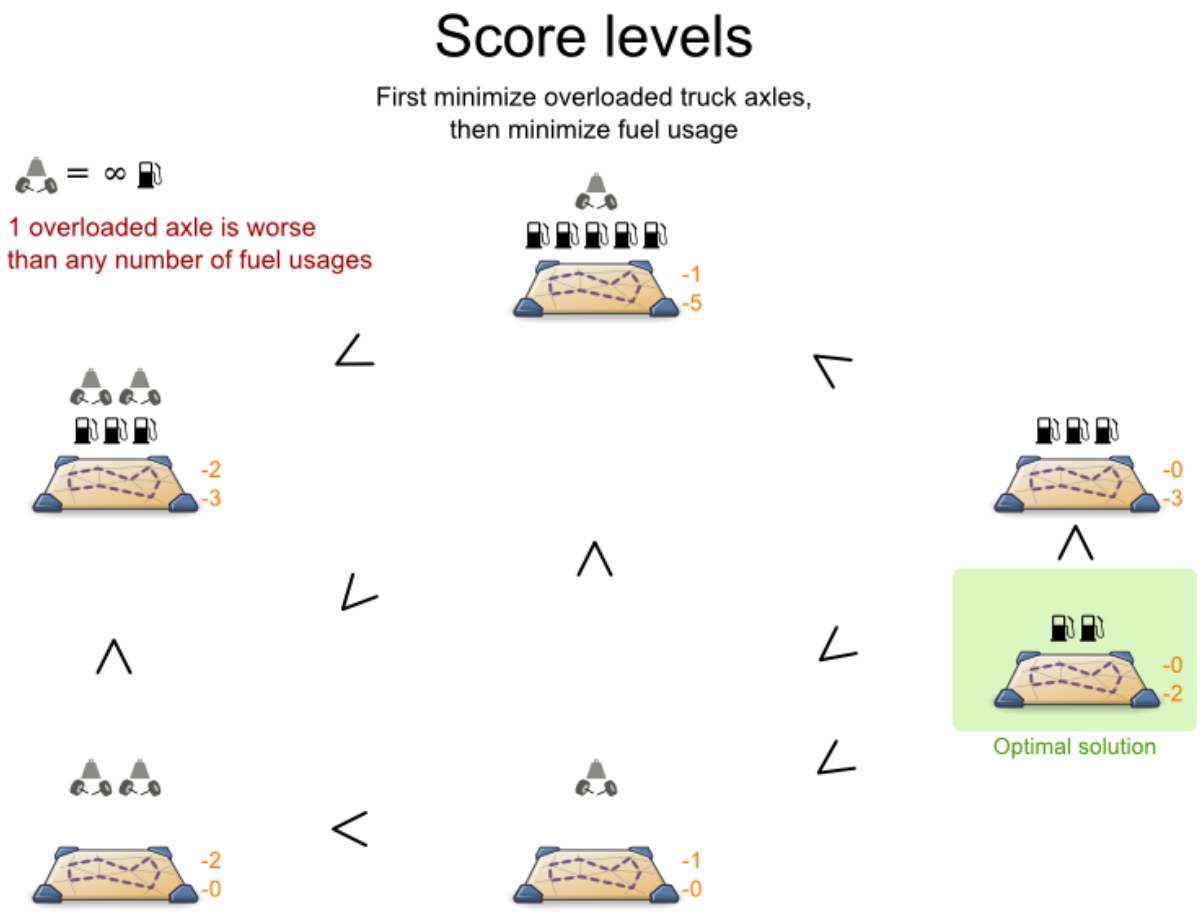
注記

さらに、**InstitutionParametrization** クラスを使用した「試験の時間割」の例で示されているように、エンドユーザーがユーザーインターフェースで制約の重みを再調整できるようにすると便利になることが多いです。

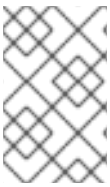
5.1.4. スコア制約レベル (ハード、ソフトなど)

スコア制約を何回違反しても、そのスコア制約のスコアが他の制約よりも高くなる場合があります。このような場合には、スコア制約のレベルが異なります。たとえば、看護師は、(物理的に制約があり) 2 つのシフトを同時に勤務できないため、これはすべての満足度制約よりも勝ります。

多くのユースケースでは、ハードとソフトの 2 つのスコアレベルのみを使用します。2 つのスコアは、辞書式に比較されます。つまり、最初のスコアレベルが先に比較され、そのスコアレベルが異なると、他のスコアレベルは無視されます。たとえば、「ハード制約 0 件 + ソフト制約 100 万件」違反したスコアの方が、「ハード制約を 1 件 + ソフト制約を 0 件」違反したスコアよりも高くなります。



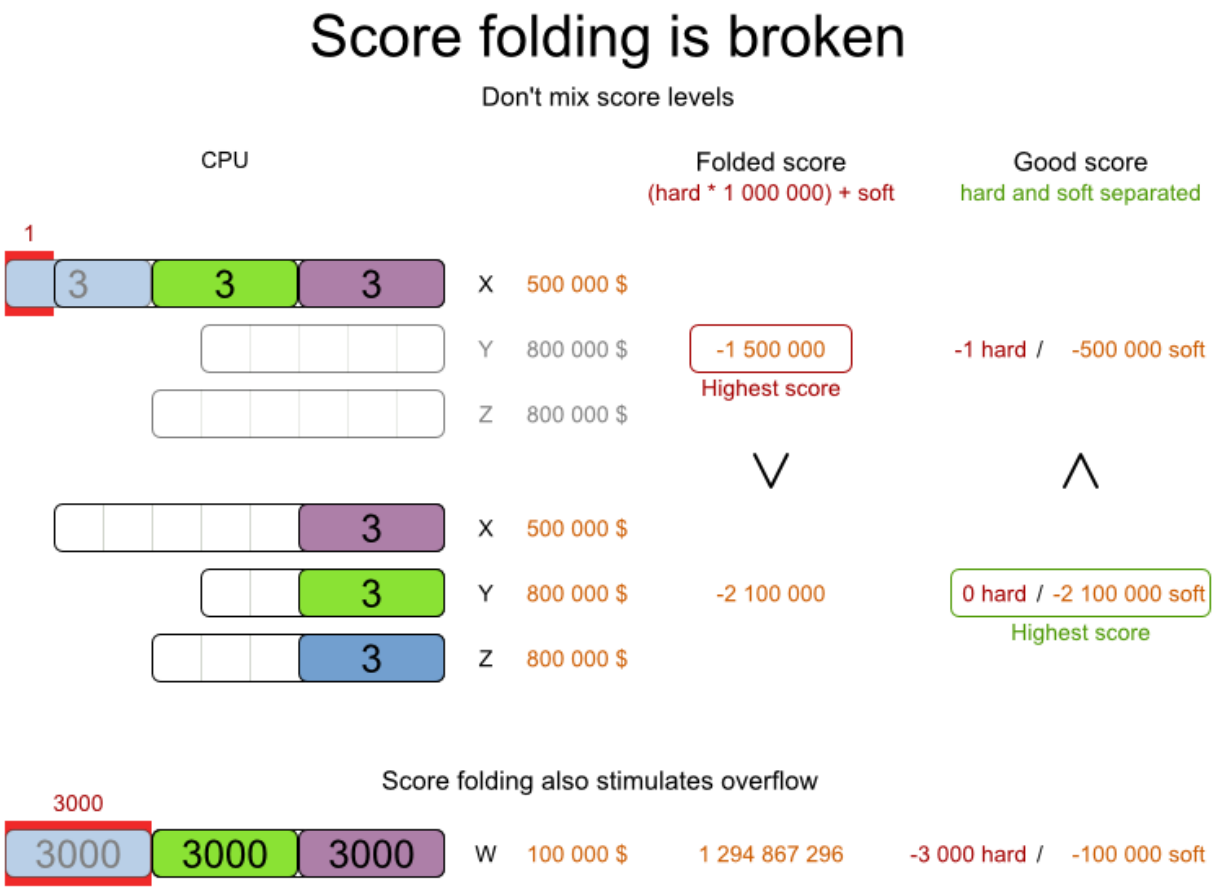
「ハードレベルとソフトレベル」など、制約スコアレベルが 2 つ (以上) 存在する場合、そのスコアはハード制約に違反していないときにのみ評価されます。

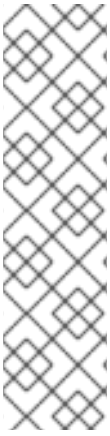


注記

デフォルトでは、Planner は、常に全プランニング変数に計画値を割り当てます。実現可能解がないと、最善解には到達しません。代わりに、プランニングエンティティーの一部を割り当てないようにするには、過制約の計画 を適用します。

制約ごとに、スコアレベル、スコアの重み、スコア符号を選択する必要があります。たとえば、**-1soft** の場合、スコアレベルは **soft**、重みは **1**、符号は負になります。実際のビジネスで別のスコアレベルを希望する場合は、制約の重みを大きくしないでください。**score folding** (スコアフォールディング) と呼ばれるハックに違反します。





注記

ビジネスの条件によりハード制約には違反できないため、すべての重みを同じにする (つまり、重みを重要視しない) 場合も注意は必要です。固有のデータセットに実行可能解が存在しなくても、少なくとも実現不可解を使用して、ビジネスに欠けているビジネスリソースがいくつあるかを推測することはできるため、重みがすべて同じというのは正しくありません。たとえば、「クラウドのバランス」サンプルにおける「新たに購入すべきコンピューターの数」などが例としてあげられます。

さらに、スコアトラップ が作成される可能性が高くなります。たとえば、「クラウドのバランス」において、コンピューター で プロセス を処理する際、CPU が 7 基分少ない場合は、CPU が 1 基分のみ足りないときの 7 倍の重みを課す必要があります。

スコアレベルは、3 つ以上サポートされます。たとえば、ある企業が、収益は従業員の満足度より重要である (あるいはその反対) と決定しても、どちらも、物理的な実際の制約よりも優先されることはないようにします。

注記

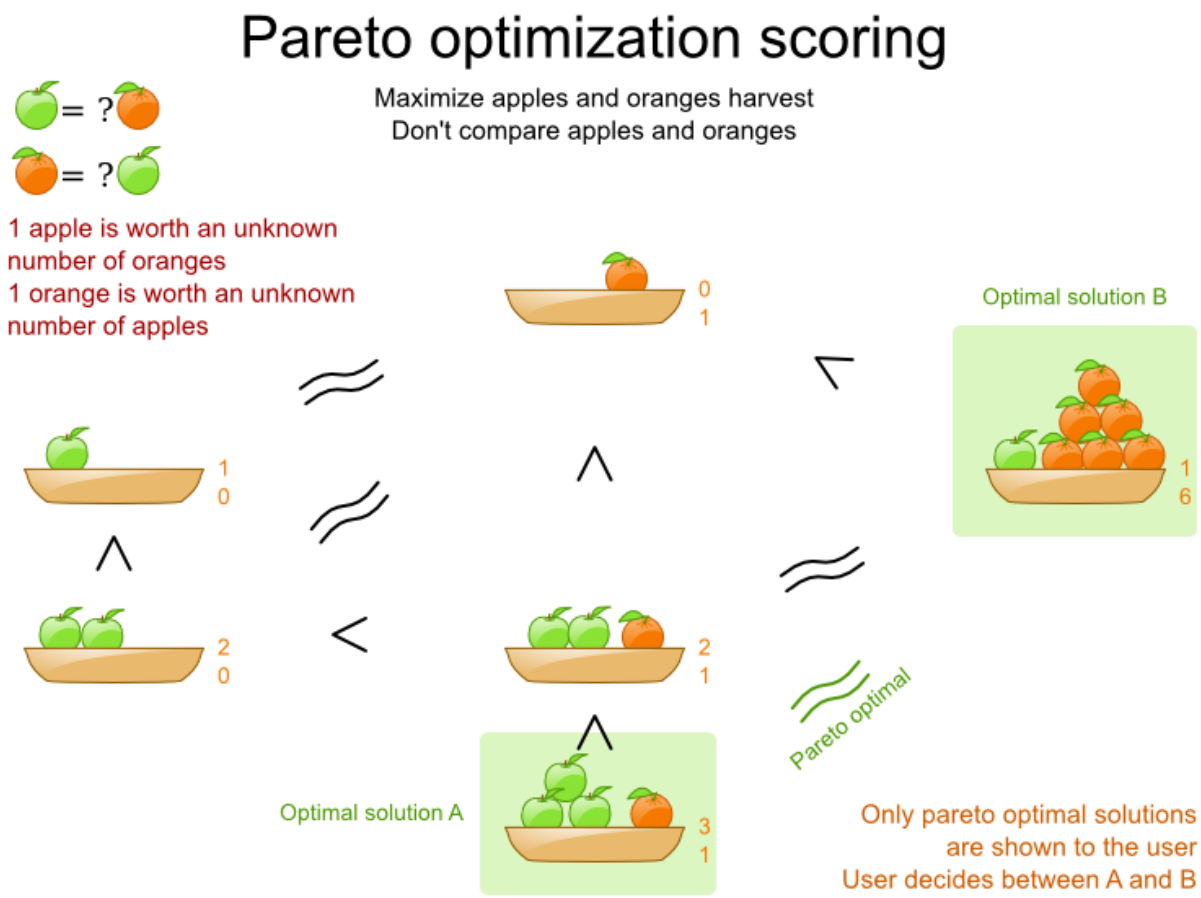
(Planner は多くのスコアレベルを処理できますが) 公平さまたは負荷分散をモデル化するのに、スコアレベルを多数使用する必要はありません 。

5.1.5. パレートスコアリング (多目的最適化スコアリング)

(多目的最適化としても知られている) パレート最適化のユースケースは、他と比べてあまり一般的ではありません。パレートスコアリングでは、スコア制約のスコアレベルは同じですが、相互に重みが課さ

れているわけではありません。2 つのスコアを比較する際に、各スコア制約が個別に比較され、最も優勢なスコア制約を持つスコアが勝ちます。パレートスコアリングは、スコアレベルやスコア制約の重みと組み合わせることができます。

正の制約を使用する例を見てみましょう。ここでは、りんごとオレンジを最も多く取得することを目的にします。りんごとオレンジは比較できないので、それぞれに重みを課します。ただし、比較はできなくても、りんご 2 個はりんご 1 個に勝っていることは記述できます。同様に、りんご 2 個とオレンジ 1 個は、オレンジ 1 個だけよりも勝っていることも記述できます。最終的に (スコアを同じに宣言する時点)で スコアを比較できなくても、最適なスコアセットを特定できます。このようなスコアを、パレート最適と呼びます。



スコアが同じと見なされることが非常に多くなり、Planner が探し出した複数の最善解 (同等のスコア) から手動で選択できるようになっています。上の例で、ユーザーはソリューション A (りんご 3 個とオレンジ 1 個) とソリューション B (りんご 1 個とオレンジ 6 個) の中から選択する必要があります。Planner が、りんごが多いソリューション、オレンジが多いソリューション、りんごとオレンジの組み合わせがより適切なソリューション (りんご 2 個とオレンジ 3 個など) など、他のソリューションを検出していないことは保証されます。

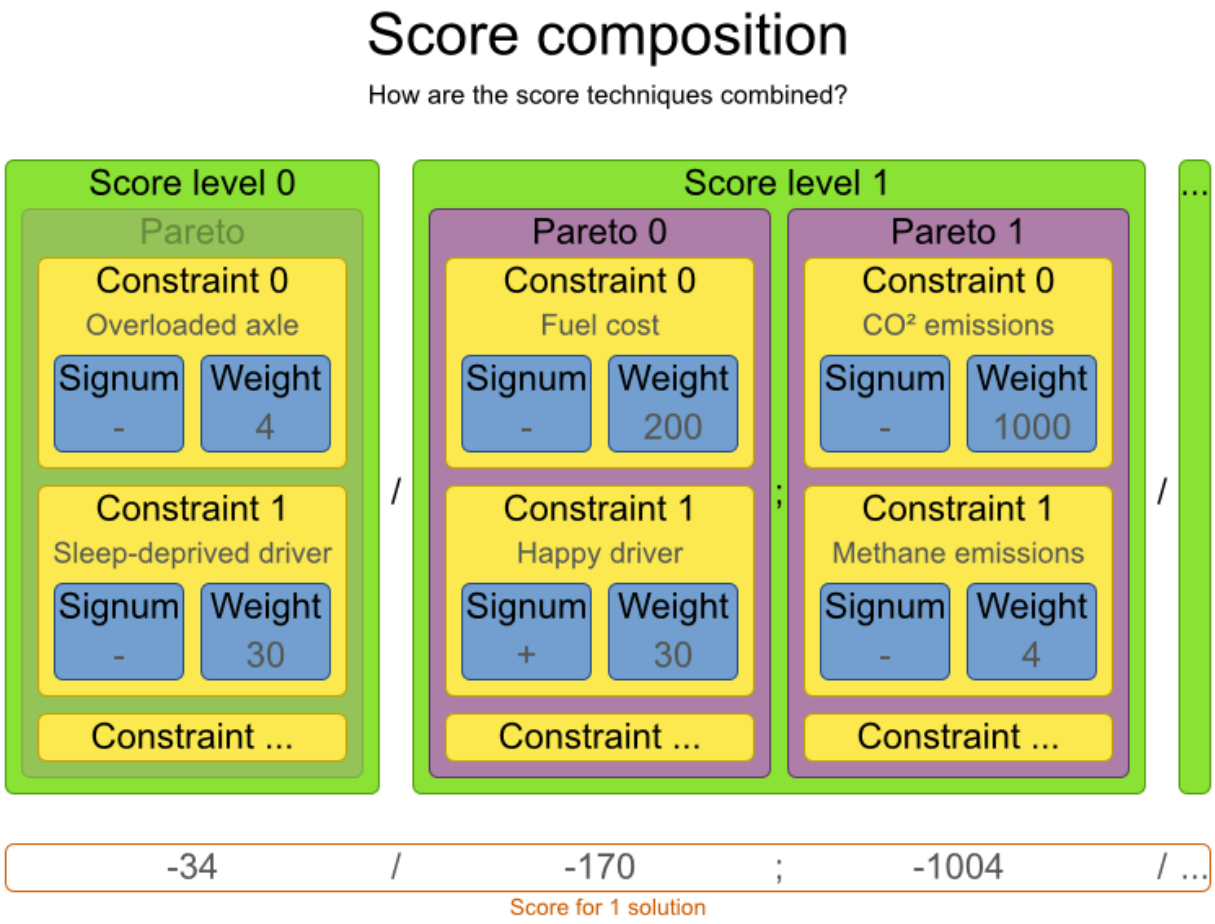
Planner にパレートスコアリングを実装するには、カスタムの ScoreDefinition およびスコアを実装 します (さらに、**BestSolutionRecaller** を置き換えます)。これは、今後のバージョンで、カスタマイズしなくても使用できるようになります。

注記

パレート スコア の **compareTo** メソッドは、パレートの比較を行うため、推移的ではありません。たとえば、りんご 2 個はりんご 1 個より優れていて、りんご 1 個とオレンジ 1 個は同じですが、りんご 2 個がオレンジ 1 個よりも優れているわけではありません (実際には同じ)。パレートの比較は、インターフェース **java.lang.Comparable** の **compareTo** メソッドのコントラクトに違反しますが、Planner のシステムは、本書でそのように記載がない限り、パレート比較でも安全に使用できます。

5.1.6. スコア手法を組み合わせて使用

これまでに説明したスコア手法はすべて、シームレスに組み合わせることができます。

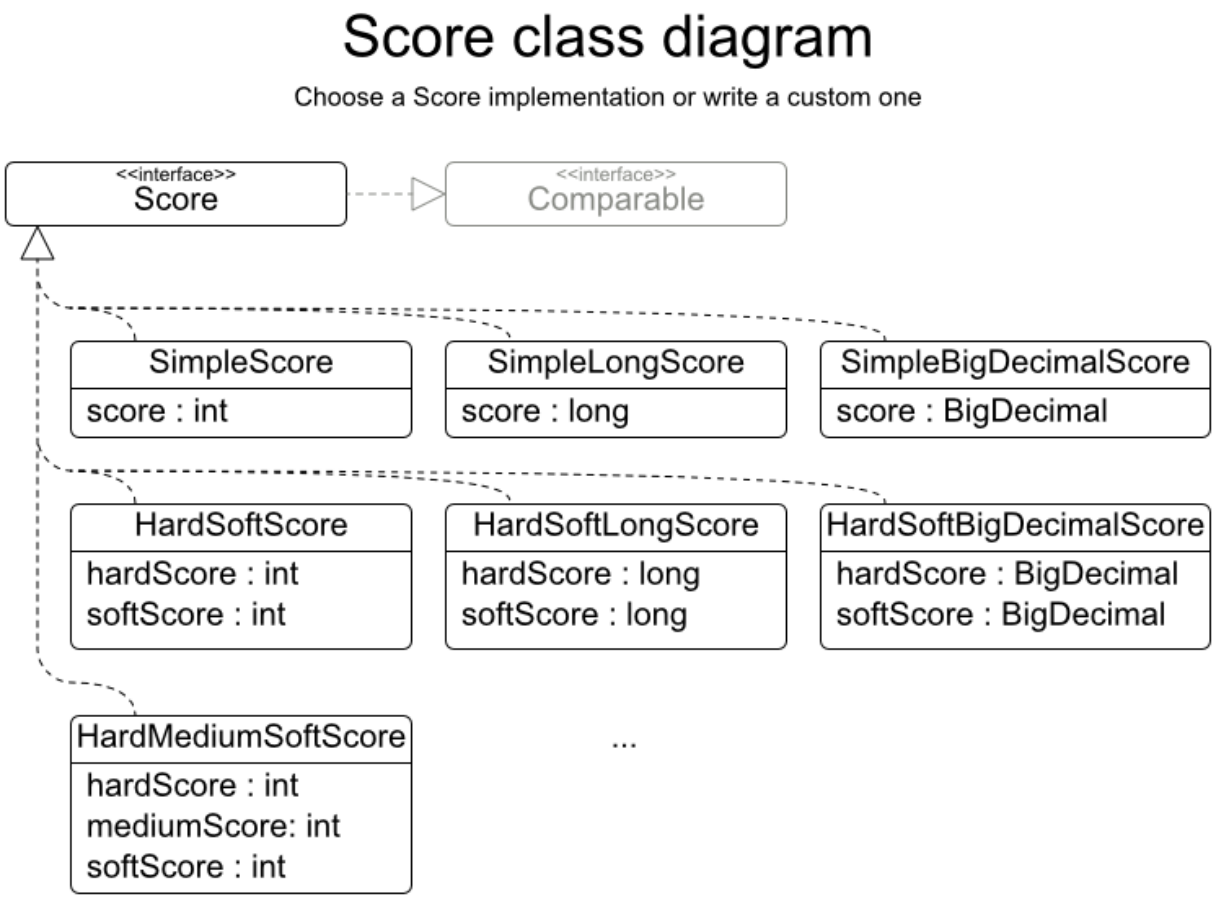


5.1.7. スコアインターフェース

スコアは、**Score** インターフェースで表現され、**Comparable** を継承します。

```
public interface Score<...> extends Comparable<...> {
    ...
}
```

使用する **Score** 実装は、ユースケースにより異なります。スコアは、1 つの **long** 値に効率よく収まらない可能性があります。Planner には、組み込みの **Score** 実装が複数含まれていますが、カスタムの **Score** も実装できます。傾向としては、組み込みの **HardSoftScore** を使用するユースケースが多くなっています。



Score 実装 (**HardSoftScore** など) は、**Solver** のランタイム中は常に同じでなければなりません。**Score** 実装は、Solver の設定で ScoreDefinition として設定されます。

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

5.1.8. スコア計算で浮動小数点を使用しない

スコア計算に **float** と **double** を使用しないようにしてください。代わりに**BigDecimal** を使用してください。

浮動小数点 (**float** および **double**) では、小数点が正しく表現されません。たとえば、**double** は **0.05** を正しく保持できず、代わりに最近似値が保持されます。したがって、特に計画問題において、浮動小数点が含まれる (加算や減算などの) 演算があると、決定が間違ってしまう可能性があります。

さらに、浮動小数点数の加算は結合的ではありません。

```
System.out.println( ((0.01 + 0.02) + 0.03) == (0.01 + (0.02 + 0.03)) ); // returns false
```

これにより スコアが正しくなくなります。

10 進数 (**BigDecimal**) ではこの問題は発生しません。

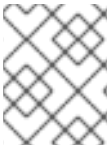
注記

BigDecimal の演算の処理は、**int**、**long** または **double** の演算に比べてはるかに遅くなります。新手法では、計算の平均数が 5 で除算されていました。

そのため、場合によっては、スコアの重みが **int** または **long** に収まるように、すべての数に 10 の倍数 (**1000** など) を掛けて使用するとよいでしょう。

5.2. スコア定義の選択

各 **Score** 実装には、**ScoreDefinition** 実装も含まれます。たとえば、**SimpleScore** は **SimpleScoreDefinition** により定義されます。



注記

データベース (JPA/Hibernate) または XML/JSON (XStream/JAXB) に スコア を適切に記述する方法は、「統合」の章を参照してください。

5.2.1. SimpleScore

SimpleScore には、**-123** などの **int** の値が含まれ、SimpleScore のスコアレベルは 1 つとなっています。

```
<scoreDirectorFactory>
  <scoreDefinitionType>SIMPLE</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

この **scoreDefinitionType** には以下が設定できます。

- **SIMPLE_LONG**: **int** の値ではなく、**long** の値を含む **SimpleLongScore** を使用します。
- **SIMPLE_DOUBLE**: **int** の値ではなく、**double** の値を含む **SimpleDoubleScore** を使用します。これを使用することは推奨されていません。
- **SIMPLE_BIG_DECIMAL**: **int** 値ではなく、**BigDecimal** 値を含む **SimpleBigDecimalScore** を使用します。

5.2.2. HardSoftScore (推奨)

HardSoftScore には、**-123hard/-456soft** など、ハード **int** 値や ソフト **int** 値が含まれます。HardSoftScore のスコアレベルは 2 つ (ハードとソフト) です。

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

この **scoreDefinitionType** には以下が設定できます。

- **HARD_SOFT_LONG**: **int** 値ではなく、**long** 値を含む **HardSoftLongScore** を使用します。
- **HARD_SOFT_DOUBLE**: **int** 値ではなく、**double** 値を含む **HardSoftDoubleScore** を使用します。これを使用することは推奨されていません。
- **HARD_SOFT_BIG_DECIMAL**: **int** 値ではなく、**BigDecimal** 値を含む **HardSoftBigDecimalScore** を使用します。

5.2.3. HardMediumSoftScore

-123hard/-456medium/-789soft など、ハード **int** 値、ミディアム **int** 値、ソフト **int** 値を含む **HardMediumSoftScore**。HardMediumSoftScore には、スコアレベルは 3 つ (ハード、ミディアム、ソフト) になります。

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_MEDIUM_SOFT</scoreDefinitionType>
  ...
</scoreDirectorFactory>
```

この **scoreDefinitionType** には以下が設定できます。

- **HARD_MEDIUM_SOFT_LONG**: **int** 値ではなく、**long** の値を含む **HardMediumSoftLongScore** を使用します。

5.2.4. BendableScore

BendableScore では、スコアレベルの数を設定できます。ハードレベル 2 つ、ソフトレベル 3 つなど、ハード **int** 値の配列と、ソフト **int** 値の配列を含みます。スコアは、**-123/-456/-789/-012/-345** などになります。

```
<scoreDirectorFactory>
  <scoreDefinitionType>BENDABLE</scoreDefinitionType>
  <bendableHardLevelsSize>2</bendableHardLevelsSize>
  <bendableSoftLevelsSize>3</bendableSoftLevelsSize>
  ...
</scoreDirectorFactory>
```

ハードおよびソフトのスコアレベルの数は、設定時に指定する必要があります。問題解決時に変更することはできません。

この **scoreDefinitionType** には以下が設定できます。

- **BENDABLE_LONG**: **int** 値ではなく、**long** を含む **BendableLongScore** を使用します。
- **BENDABLE_BIG_DECIMAL**: **int** 値ではなく、**BigDecimal** を含む **BendableBigDecimalScore** を使用します。

5.2.5. カスタムスコアの実装

ScoreDefinition インターフェースは、スコア表現を定義します。

カスタムの スコア を実装するために、カスタムの **ScoreDefinition**を実装する必要があります。**AbstractScoreDefinition** (できれば **HardSoftScoreDefinition** をコピーアンドペーストして) を継承します。

次に、**SolverConfig.xml** でカスタムの **ScoreDefinition** をフックします。

```
<scoreDirectorFactory>
  <scoreDefinitionClass>...MyScoreDefinition</scoreDefinitionClass>
  ...
</scoreDirectorFactory>
```

JPA/Hibernate や XStream などとシームレスに組み合わせるには、グルーコードを記述する必要があります。

5.3. スコア計算

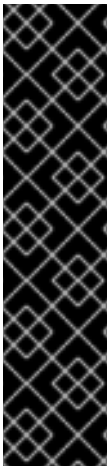
5.3.1. スコア計算のタイプ

ソリューション の スコア を計算する方法は複数あります。

- **Easy Java** のスコア計算: Java メソッドを 1 つ実装します。
- **Java** インクリメント演算子によるスコア計算: 複数の Java メソッドを実装します。
- **Drools** スコア計算 (推奨): スコアルールを実装します。

スコア算出タイプはすべてのスコア定義を使用できます。たとえば、Easy Java スコア計算は **HardSoftScore** を出力します。

スコア計算のタイプはすべてオブジェクト指向型で、既存の Java コードを再利用することができます。



重要

スコア計算は読み取り専用にし、プランニングエンティティーまたは問題ファクトを変更しないようにしてください。Drools スコアルールの RHS 内において、プランニングエンティティーでセッターメソッドを呼び出さないようにしてください。これは、論理的に挿入された オブジェクトには適用されません。このオブジェクトは、論理的に挿入されているスコアルールにより変更できます。

(environmentMode アサーション が有効になっていない限り) Planner が ソリューションのスコアを推測できる場合には、ソリューションが再計算されません。たとえば、Move が勝利ステップの前に実行し元に戻ったため、勝利ステップの後にスコアを計算する必要はありません。そのため、スコア計算中に適用されたこのような変更が実際に行われる保証はありません。

5.3.2. Easy Java スコア計算

Java でスコア計算を簡単に実装する方法には、以下の特徴があります。

- メリット:
  - Plain old Java (POJO): 学習曲線がない
  - 既存のコードベースやレガシーシステムにスコア計算を委譲する機会がある
- デメリット:
  - 処理速度が遅く、スケーラビリティーが低い
    - インクリメンタルスコアの計算 がないため

EasyScoreCalculator インターフェースのメソッドを 1 つだけ実装します。

```
public interface EasyScoreCalculator<Sol extends Solution> {

    Score calculateScore(Sol solution);

}
```

たとえば、N クィーンの例では以下のようになります。

```
public class NQueensEasyScoreCalculator implements EasyScoreCalculator<NQueens> {

    public SimpleScore calculateScore(NQueens nQueens) {
        int n = nQueens.getN();
        List<Queen> queenList = nQueens.getQueenList();

        int score = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                Queen leftQueen = queenList.get(i);
```

```
                Queen rightQueen = queenList.get(j);
                if (leftQueen.getRow() != null && rightQueen.getRow() != null) {
                    if (leftQueen.getRowIndex() == rightQueen.getRowIndex()) {
                        score--;
                    }
                    if (leftQueen.getAscendingDiagonalIndex() == rightQueen.getAscendingDiagonalIndex())
{
                        score--;
                    }
                    if (leftQueen.getDescendingDiagonalIndex() ==
rightQueen.getDescendingDiagonalIndex()) {
                        score--;
                    }
                }
            }
        }
        return SimpleScore.valueOf(score);
    }

}
```

Solver の設定では以下のように設定します。

```
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>

<easyScoreCalculatorClass>org.optaplanner.examples.nqueens.solver.score.NQueensEasyScoreCalc
ulator</easyScoreCalculatorClass>
  </scoreDirectorFactory>
```

または、ランタイム時に **EasyScoreCalculator** インスタンスを構築して、Programmatic API で設定します。

```
solverFactory.getSolverConfig().getScoreDirectorFactoryConfig.setEasyScoreCalculator(easyScoreCalc
ulator);
```

5.3.3. Java インクリメント演算子によるスコア計算

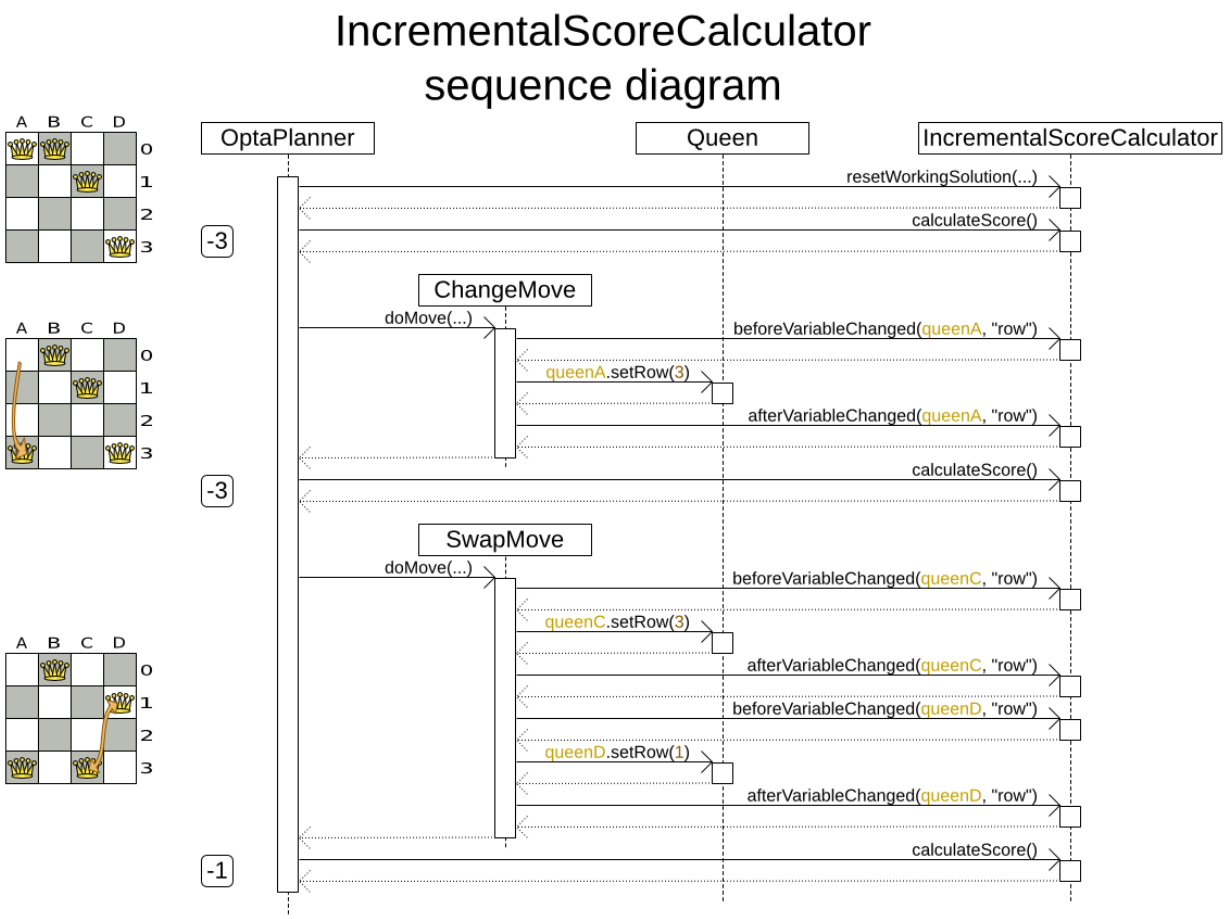
Java でインクリメンタルにスコア計算を実装する方法には、以下の特徴があります。

- メリット:
  - 非常に速く、スケーラビリティーが高い
    - 正しく実装されている場合には、最速である
- デメリット:
  - 記述が難しい
    - スケーラブルな実装はマッピング、インデックスなどを多用する (Drools ルールエンジンで対応できる内容)。
    - これらのパフォーマンス最適化をすべて、学習、設計、記述、改善する必要がある

- 読み込みが困難である
  - スコア制約を定期的に変更すると、メンテナンスコストがかさむ可能性がある

インターフェース **IncrementalScoreCalculator** の全メソッドを実装して、**AbstractIncrementalScoreCalculator** クラスを継承します。

```
public interface IncrementalScoreCalculator<Sol extends Solution> {

    void resetWorkingSolution(Sol workingSolution);

    void beforeEntityAdded(Object entity);

    void afterEntityAdded(Object entity);

    void beforeVariableChanged(Object entity, String variableName);

    void afterVariableChanged(Object entity, String variableName);

    void beforeEntityRemoved(Object entity);

    void afterEntityRemoved(Object entity);

    Score calculateScore();

}
```



たとえば、N クィーンの例では以下のようになります。

```
public class NQueensAdvancedIncrementalScoreCalculator extends
AbstractIncrementalScoreCalculator<NQueens> {

    private Map<Integer, List<Queen>> rowIndexMap;
    private Map<Integer, List<Queen>> ascendingDiagonalIndexMap;
    private Map<Integer, List<Queen>> descendingDiagonalIndexMap;

    private int score;

    public void resetWorkingSolution(NQueens nQueens) {
        int n = nQueens.getN();
        rowIndexMap = new HashMap<Integer, List<Queen>>(n);
        ascendingDiagonalIndexMap = new HashMap<Integer, List<Queen>>(n * 2);
        descendingDiagonalIndexMap = new HashMap<Integer, List<Queen>>(n * 2);
        for (int i = 0; i < n; i++) {
            rowIndexMap.put(i, new ArrayList<Queen>(n));
            ascendingDiagonalIndexMap.put(i, new ArrayList<Queen>(n));
            descendingDiagonalIndexMap.put(i, new ArrayList<Queen>(n));
            if (i != 0) {
                ascendingDiagonalIndexMap.put(n - 1 + i, new ArrayList<Queen>(n));
                descendingDiagonalIndexMap.put((-i), new ArrayList<Queen>(n));
            }
        }
        score = 0;
        for (Queen queen : nQueens.getQueenList()) {
            insert(queen);
        }
    }

    public void beforeEntityAdded(Object entity) {
        // Do nothing
    }

    public void afterEntityAdded(Object entity) {
        insert((Queen) entity);
    }

    public void beforeVariableChanged(Object entity, String variableName) {
        retract((Queen) entity);
    }

    public void afterVariableChanged(Object entity, String variableName) {
        insert((Queen) entity);
    }

    public void beforeEntityRemoved(Object entity) {
        retract((Queen) entity);
    }

    public void afterEntityRemoved(Object entity) {
        // Do nothing
    }

    private void insert(Queen queen) {
        Row row = queen.getRow();
        if (row != null) {
```

```
            int rowIndex = queen.getRowIndex();
            List<Queen> rowIndexList = rowIndexMap.get(rowIndex);
            score -= rowIndexList.size();
            rowIndexList.add(queen);
            List<Queen> ascendingDiagonalIndexList =
ascendingDiagonalIndexMap.get(queen.getAscendingDiagonalIndex());
            score -= ascendingDiagonalIndexList.size();
            ascendingDiagonalIndexList.add(queen);
            List<Queen> descendingDiagonalIndexList =
descendingDiagonalIndexMap.get(queen.getDescendingDiagonalIndex());
            score -= descendingDiagonalIndexList.size();
            descendingDiagonalIndexList.add(queen);
        }
    }

    private void retract(Queen queen) {
        Row row = queen.getRow();
        if (row != null) {
            List<Queen> rowIndexList = rowIndexMap.get(queen.getRowIndex());
            rowIndexList.remove(queen);
            score += rowIndexList.size();
            List<Queen> ascendingDiagonalIndexList =
ascendingDiagonalIndexMap.get(queen.getAscendingDiagonalIndex());
            ascendingDiagonalIndexList.remove(queen);
            score += ascendingDiagonalIndexList.size();
            List<Queen> descendingDiagonalIndexList =
descendingDiagonalIndexMap.get(queen.getDescendingDiagonalIndex());
            descendingDiagonalIndexList.remove(queen);
            score += descendingDiagonalIndexList.size();
        }
    }

    public SimpleScore calculateScore() {
        return SimpleScore.valueOf(score);
    }

}
```

Solver の設定では以下のように設定します。

```
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>

<incrementalScoreCalculatorClass>org.optaplanner.examples.nqueens.solver.score.NQueensAdvance
dIncrementalScoreCalculator</incrementalScoreCalculatorClass>
  </scoreDirectorFactory>
```

オプションで、**IncrementalScoreCalculator** が **FAST_ASSERT** または **FULL_ASSERT** **environmentMode** に折り畳まれている場合に、**ScoreDirector.getConstraintMatchTotals()** でスコアを説明するか、より良い出力を取得するには、**ConstraintMatchAwareIncrementalScoreCalculator** インターフェースも実装します。

```
public interface ConstraintMatchAwareIncrementalScoreCalculator<Sol extends Solution> {

    void resetWorkingSolution(Sol workingSolution, boolean constraintMatchEnabled);
```

```
    Collection<ConstraintMatchTotal> getConstraintMatchTotals();

}
```

5.3.4. Drools スコア計算

5.3.4.1. 概要

Drools ルールエンジンを使用してスコア計算を実装します。すべてのスコア制約を、1 つまたは複数のスコアルールとして記述します。

- メリット:
  - 追加設定せずにスコアのインクリメンタル計算ができる
    - 多くの DRL 構文は前向き連鎖を使用するので、追加のコードなしでインクリメントの計算が可能である。
  - スコア制約が別のルールに分かれている
    - 既存のスコアルールを簡単に追加または編集できる
  - スコア制約を柔軟に増やすことができる
    - 方法: ディションテーブルで以下を定義する
      - Excel (XLS) スプレッドシート
      - KIE Workbench WebUI
    - DSL で自然言語に変換できる
    - KIE Workbench リポジトリーで格納し、リリースできる
  - 今後のバージョンでも、追加設定せずにパフォーマンスの最適化ができる
    - どのリリースでも、Drools ルールエンジンの速度が向上する傾向にある
- デメリット:
  - DRL の学習曲線
  - DRL の使用
    - 多言語の使用が懸念されるため、組織によっては DRL などの新しい言語の使用が禁止される場合があります。

5.3.4.2. Drools スコアルールの設定

スコアルールの配置場所を定義する方法は複数存在します。

5.3.4.2.1. クラスパスの scoreDrl リソース

これは、単純な方法です。スコアルールを、クラスパスリソースとして提供される DRL ファイルに配置します。Solver の設定の **<scoreDrl>** 要素に、スコアルールの DRL ファイルを追加します。

```
<scoreDirectorFactory>
  <scoreDefinitionType>...</scoreDefinitionType>
  <scoreDrl>org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</scoreDrl>
</scoreDirectorFactory>
```

通常のプロジェクト (Maven ディレクトリー構造に準拠) では、DRL ファイルは **$PROJECT_DIR/src/main/resources/org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl** (WAR プロジェクトの場合も同様) に配置されます。

注記

<scoreDrl> 要素には、**ClassLoader.getResource(String)** で定義されたクラスパスリソースが必要です。ファイル、URL、Webapp リソースは受け入れられません。以下を参照して、ファイル を使用してください。

スコアルールが複数の DRL ファイルに分かれている場合は、**<scoreDrl>** 要素を複数追加します。

オプションで、Drools 設定プロパティーも設定できます (ただし、後方互換性の問題に注意してください)。

```
<scoreDirectorFactory>
  ...
  <scoreDrl>org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</scoreDrl>
  <kieBaseConfigurationProperties>
    <drools.equalityBehavior>...</drools.equalityBehavior>
  </kieBaseConfigurationProperties>
</scoreDirectorFactory>
```

5.3.4.2.2. scoreDrlFile

ローカルのファイルシステムで ファイル を使用するには、クラスパスリソースの代わりに、Solver 設定の **<scoreDrlFile>** 要素に、スコアルールの DRL ファイルを追加します。

```
<scoreDirectorFactory>
  <scoreDefinitionType>...</scoreDefinitionType>
  <scoreDrlFile>/home/ge0ffrey/tmp/nQueensScoreRules.drl</scoreDrlFile>
</scoreDirectorFactory>
```

警告

移植性の理由から、ファイルよりもクラスパスリソースが推奨されます。あるコンピューターに構築されているアプリケーションを別のコンピューターで使用する場合に、ファイルの場所が異なる可能性があります。さらに、オペレーティングシステムが異なる場合に、移植可能なファイルパスを選択するのは困難です。

スコアルールが複数の DRL ファイルに分かれている場合は、**<scoreDrlFile>** 要素を複数追加します。

5.3.4.2.3. Maven リポジトリーからの KJAR の ksessionName

これを使用することで、Workbench で定義されたスコアルールを使用するか、KJAR を構築して実行サーバーにデプロイすることができます。スコアルールも Solver 設定も KJAR のリソースです。クライアントは、ローカルクラスパス、ローカルの Maven リポジトリー、またはリモートの Maven リポジトリーから KJAR を取得できます。

スコアルールは DRL ファイルに配置されていますが、**KieContainer** は **META-INF/kmodule.xml** ファイル経由でこのファイルを検索します。

```
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="nQueensKbase" packages="org.optaplanner.examples.nqueens.solver">
    <ksession name="nQueensKsession"/>
  </kbase>
</kmodule>
```

上記の kmodule で、**org.optaplanner.examples.nqueens.solver** パッケージの DRL ファイルをすべて取得します。また、kbase は別の kbase を継承することも可能です。

Solver の設定の **<ksessionName>** 要素に ksession 名を追加します。

```
  <scoreDirectorFactory>
    <scoreDefinitionType>...</scoreDefinitionType>
    <ksessionName>nQueensKsession</ksessionName>
  </scoreDirectorFactory>
```

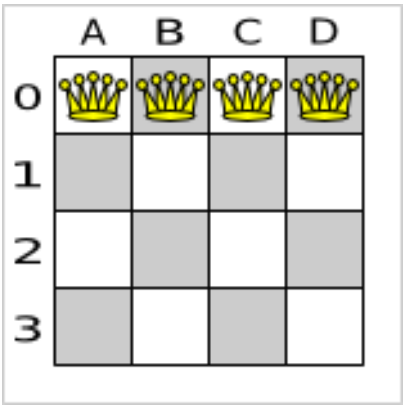
このアプローチを使用する場合には、**SolverFactory.createFromKieContainerXmlResource(...)** メソッドを使用して **SolverFactory** を構築する必要があります。

5.3.4.3. スコアルールの実装

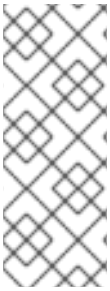
以下に、DRL ファイルにスコアルールとして実装されたスコア制約の例を紹介します。

```
rule "multipleQueensHorizontal"
    when
        Queen($id : id, row != null, $i : rowIndex)
        Queen(id > $id, rowIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end
```

このスコアルールは、**rowIndex** が同じクィーン 2 つごとに 1 回、トリガーされます。クィーンが A と B の 2 つある場合に、(A, B) にだけトリガーし、(B, A)、(A, A) または (B, B) にはトリガーされないようにするには、**(id > $id)** の条件が必要です。「4 個のクィーン」の以下のソリューションを使用して、スコアルールについて詳しく見ていきましょう。



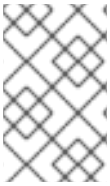
このソリューションでは、6 つ (A, B)、(A, C)、(A, D)、(B, C)、(B, D) および (C, D) に対して multipleQueensHorizontal のスコアルールが発生します。クィーンは同じ垂直線上または対角線上にないので、このソリューションのスコアは **-6** になります。なお、「4 個のクィーン」の最適解スコアは **0** です。



注記

全スコアルールは、 (論理的に挿入されたファクトを直接的または間接的に使用して) 少なくとも 1 つのプランニングエンティティークラスに関連付けます。

これは、通常の例です。考えられるソリューションが何であっても計画中に結果は変わらないため、問題ファクトにだけ関連するスコアルールを記述しても効果はありません。



注記

kcontext 変数は、Drools Expert のマジック変数です。 **scoreHolder** のメソッドは、この変数を使用しインクリメンタルスコアの計算を正しく行い、**ConstraintMatch** インスタンスを作成します。

5.3.4.4. スコアルールの重みづけ

ScoreHolder インスタンスは、**scoreHolder** のグローバルとして **KieSession** にアサートし、スコアルールで (直接的または間接的に) 更新する必要があります。

```
global SimpleScoreHolder scoreHolder;

rule "multipleQueensHorizontal"
    when
        Queen($id : id, row != null, $i : rowIndex)
        Queen(id > $id, rowIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end

// multipleQueensVertical is obsolete because it is always 0

rule "multipleQueensAscendingDiagonal"
    when
        Queen($id : id, row != null, $i : ascendingDiagonalIndex)
        Queen(id > $id, ascendingDiagonalIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
```

```
end

rule "multipleQueensDescendingDiagonal"
    when
        Queen($id : id, row != null, $i : descendingDiagonalIndex)
        Queen(id > $id, descendingDiagonalIndex == $i)
    then
        scoreHolder.addConstraintMatch(kcontext, -1);
end
```

注記

Drools ルール言語 (DRL) の詳細は、Drools ドキュメント を参照してください。

ユースケースの多くは、制約の各一致に対する重みを使用することで、制約タイプや、一致そのものに課す重みを変えます。たとえば、コースの時間割 では、授業 を、座席が 2 つ足りない 講義室 に割り当てるのは、カリキュラム に、孤立した 授業 1つを割り当てる時と同じだけマイナスと評価するように、重みを課しています。

```
global HardSoftScoreHolder scoreHolder;

// RoomCapacity: For each lecture, the number of students that attend the course must be less or
equal
// than the number of seats of all the rooms that host its lectures.
rule "roomCapacity"
    when
        $room : Room($capacity : capacity)
        $lecture : Lecture(room == $room, studentSize > $capacity, $studentSize : studentSize)
    then
        // Each student above the capacity counts as 1 point of penalty.
        scoreHolder.addSoftConstraintMatch(kcontext, ($capacity - $studentSize));
end

// CurriculumCompactness: Lectures belonging to a curriculum should be adjacent
// to each other (i.e., in consecutive periods).
// For a given curriculum we account for a violation every time there is one lecture not adjacent
// to any other lecture within the same day.
rule "curriculumCompactness"
    when
        ...
    then
        // Each isolated lecture in a curriculum counts as 2 points of penalty.
        scoreHolder.addSoftConstraintMatch(kcontext, -2);
end
```

5.3.5. InitializingScoreTrend

InitializingScoreTrend は、変数が初期化されるにつれ、(すでに初期化済みの変数は変化しませんが) スコアがどのように変化するかを指定します。このような情報があれば、最適化アルゴリズム (このような構造ヒューリスティックやしらみつぶし探索) の実行速度は速くなります。

スコア (または個別の スコアレベル) では、トレンドを指定します。

- **ANY** (デフォルト): 追加の変数を初期化すると、スコアが正または負に変化する可能性があります。パフォーマンスが向上することはありません。
- **ONLY_UP** (あまり使用されない): 追加の変数を初期化すると、スコアを正にだけ変更します。つまり、以下を意味します。
  - 正の制約しかない
  - 次の変数を初期化しても、前の初期化変数と一致した正の制約が一致しなくなるように、元に戻すことはできない
- **ONLY_DOWN**: 追加の変数を初期化すると、スコアを負にだけ変更します。つまり、以下を意味します。
  - 負の制約しかない
  - 次の変数を初期化しても、前の初期化変数と一致した負の制約の一致が一致しなくなるように、元に戻すことはできない

負の制約しかないユースケースが多く、中でも、スコアが下がるだけの **InitializingScoreTrend** を使用するユースケースが多くなっています。

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  <scoreDrl>.../cloudBalancingScoreRules.drl</scoreDrl>
  <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
</scoreDirectorFactory>
```

または、各スコアレベルのトレンドを個別に指定することもできます。

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  <scoreDrl>.../cloudBalancingScoreRules.drl</scoreDrl>
  <initializingScoreTrend>ONLY_DOWN/ONLY_DOWN</initializingScoreTrend>
</scoreDirectorFactory>
```

5.3.6. 無効なスコアの検出

FULL_ASSERT (または **FAST_ASSERT**) に **environmentMode** を配置して、インクリメンタルスコアの計算 の破損を検出します。詳しい情報は、「**environmentMode**」に関するセクションを参照してください。ただし、スコア計算プログラムでは、実際にビジネスが希望する内容に、スコア制約が実装されているかどうかの確認はできません。

インクリメンタルスコア計算プログラムのコードには、記述やレビューが困難な箇所があります。異なる実装 (例: **EasyScoreCalculator**) を使用して精度をアサートし、**environmentMode** でトリガーされるアサーションを実行します。異なる実装を、**assertionScoreDirectorFactory** として設定します。

```
<environmentMode>FAST_ASSERT</environmentMode>
...
<scoreDirectorFactory>
  <scoreDefinitionType>...</scoreDefinitionType>
  <scoreDrl>org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</scoreDrl>
  <assertionScoreDirectorFactory>

<easyScoreCalculatorClass>org.optaplanner.examples.nqueens.solver.score.NQueensEasyScoreCalc
```

```
ulator</easyScoreCalculatorClass>
    </assertionScoreDirectorFactory>
  </scoreDirectorFactory>
```

このように設定すると、**scoreDrl** が **EasyScoreCalculator** で検証されます。

5.4. スコア計算のパフォーマンスのヒント

5.4.1. 概要

Solver は、通常、その実行時間のほとんどをスコア計算に費やします (最も深いループで呼び出されます)。スコア計算が速いと、同じアルゴリズムを使用した場合に、同じ解 (ソリューション) をより短い時間で返します。一般的には、最善解が同じ時間内に返されます。

5.4.2. 秒あたりの平均計算数

問題の解決後、**Solver** が 秒あたりの平均計算数 をログに記録します。これは、スコア計算以外の実行時間も含まれていますが、問題データセットの問題スケールにより異なるため、スコア計算のパフォーマンスを測定するのに適しています。一般的に、スケールの高い問題の場合でも、**EasyScoreCalculator** の使用時以外は、**1000** よりも高くなります。

重要

スコア計算を改善するには、最高スコアを上げるのではなく、秒あたりの平均計算数を上げることに焦点を当てます。スコア計算が大幅に改善しても、アルゴリズムがローカルまたはグローバルの最適条件に固定され、最高スコアが全く、またはほぼ改善されない場合もあります。計算数を監視している場合、スコア計算の改善は非常に簡単に確認できます。

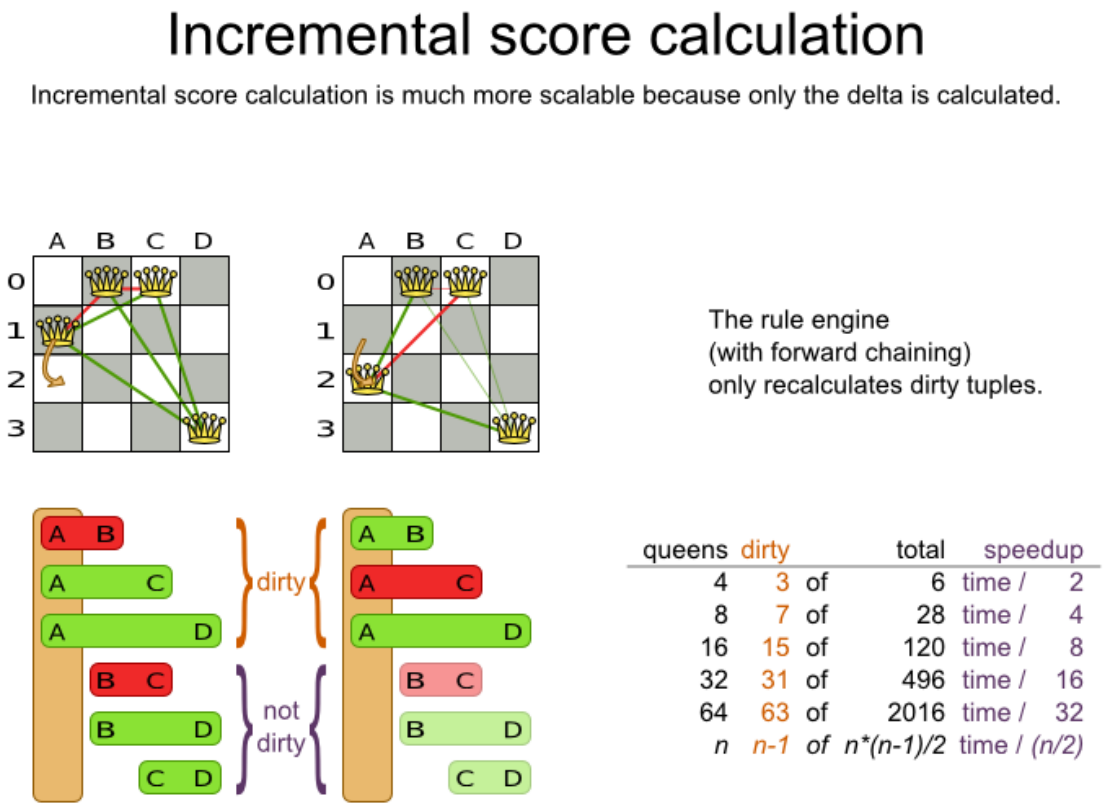
さらに、計算数を監視すると、スコア制約を追加/削除できるだけでなく、元の計算数と比較することも可能です。最高スコアと元のスコアの比較は、りんごとオレンジを比較しているようなものなので、正しくありません。

5.4.3. インクリメンタルスコア計算 (差分)

ソリューション が変化したときのインクリメンタルスコアの計算 (差分ベースのスコア計算) は、すべてのソリューションを評価する全スコアを再計算するのではなく、以前の状態の差分を計算して新しいスコア を検出します。

たとえば、クィーン A を行 **1** から **2** に移動したときは、クィーン B と C はどちらも変化しないため、B と C が相互に攻撃できるかどうかはチェックしません。

図5.1 クィーン 4 個のパズルにおけるインクリメンタルスコアの計算



これは、パフォーマンスやスケーラビリティーが大幅に改善されます。Drools スコア計算では、複雑なインクリメンタルスコアの計算アルゴリズムを記述する必要はなく、スケーラビリティーが大幅に増えます。Drools ルールエンジンに困難な作業を任せるだけです。

時間の短縮は計画問題のサイズ (n) と相対的で、インクリメンタルスコアの計算のスケーラビリティーをさらに高めることができる点に注意してください。

5.4.4. スコア計算時のリモートサービスを呼び出さない

スコア計算でリモートサービスを呼び出さないようにしてください (**EasyScoreCalculator** とレガシーシステムをブリッジングしている場合を除きます)。ネットワークにレイテンシーがあると、スコア計算のパフォーマンスを低下させます。可能な場合は、これらのリモートサービスの結果をキャッシュします。

Solver の起動時に制約の一部が計算され、解決中に変化しない場合には、キャッシュされた問題ファクト に変換します。

5.4.5. 無意味な制約

特定の制約に決して違反しない (または常に違反する) ことが分かっている場合には、それをスコア制約に記述する必要はありません。たとえば、N クィーンの例では、スコア計算で、複数のクィーンが同じ列を占有するかどうかは確認しません。これは、クィーン の 列 は絶対に変化せず、ソリューションは、必ず別の 列 にある各 クィーン で開始するためです。

注記

これは使用しすぎないようにしてください。特定の制約をデータセットが使用し、別のデータセットでは使用されていない場合には、できるだけ速く、その制約から結果を返すようにします。データセットをベースに、スコア計算を動的に変更する必要はありません。

5.4.6. 組み込みのハード制約

ハード制約を実装する代わりに、埋め込むこともできます。たとえば、授業 A は 講義室 X に割り当てるべきではありませんが、ソリューションで **ValueRangeProvider** を使用するため、**Solver** は 講義室 X にも割り当てようとします (ただし、割り当ててもハード制約に違反するだけです)。プランニングエンティティー上の ValueRangeProvider または フィルターセクション を使用して、授業 A が X 以外の講義室 にのみ割り当てられるようにします。

これにより、実行できないソリューションを多くの最適化アルゴリズムが評価する時間が短縮されるため、スコア計算が速くなるだけでなく、ユースケースによってはパフォーマンスが向上する場合があります。ただし、一般的には、短期的には利点があっても長期的には悪影響となるため、これは優れたアイデアではありません。

- 多くの最適化アルゴリズムは、プランニングエンティティーを変更する時にハード制約に自由に違反できるため、局所最適条件を回避することができます。
- どちらの実装アプローチも、各ドキュメントで説明されているように制約 (機能の互換性、自動パフォーマンス最適化の無効化) があります。

5.4.7. スコア計算のパフォーマンスに関する他のヒント

- スコア計算が正しい **Number** 型で行われることを確認します。**int** 値を合計する場合には、Drools で **double** を使用して合計すると所要時間が長くなるため、double を使用しないようにしてください。
- パフォーマンスが最適になるように、サーバーモード (**java -server**) を常に使用するようにしてください。サーバーモードをオンにすることで、パフォーマンスの改善が 50% 見られました。
- また、最新の Java バージョンを使用してください。Java 1.5 から 1.6 に切り替えることで、パフォーマンスの改善が 30% 見られました。
- 成熟していない最適化は問題の原因になる可能性があることを忘れないようにしてください。設定ベースの調節ができるように、設計が柔軟に保たれていることを確認してください。

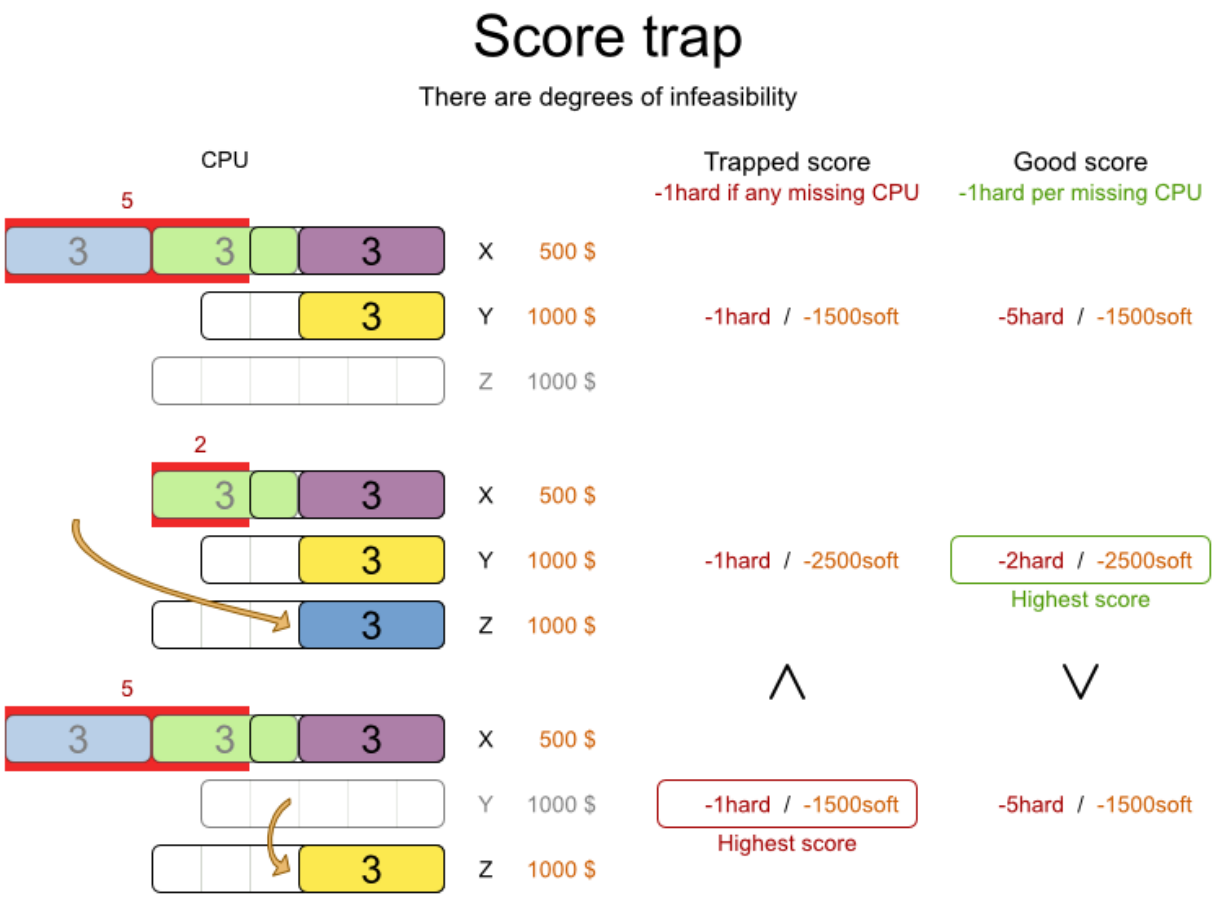
5.4.8. スコアトラップ

スコア制約がスコアトラップの原因になっていないことを確認してください。スコア制約がトラップされている場合には、重みを変えるほうが簡単であっても、異なる制約一致に同じ重みを使用しています。制約一致を効率的にまとめて、対象の制約に、上下変動のないスコア関数を作成します。これにより、Move を複数実行して解決するか、1 つの制約の重みを下げる必要があるソリューションが発生する可能性があります。スコープトラップの例は以下のとおりです。

- テーブルごとに医師が 2 名必要ですが、一度に移動できるのは医師 1 名だけです。そのため、Solver が、医師のいないテーブルに医師を 1 名移動することは誘因にはなりません。このスコア関数では、スコア制約内で、医師が 1 名しかいないテーブルよりも、医師のいないテーブルにペナルティーを課します。
- 2 つの試験を同時に実施する必要がありますが、一度に移動できるのは 1 つの試験だけです。そ

のため、Solver が、いずれかの試験を別の時間枠に移動し、同じ Move でもう 1 つの試験を移動しないようにする必要があります。同時に両試験を移動する、粒度の粗い Move を追加します。

たとえば、以下のようなスコアトラップを考えてみてください。負荷の高いコンピューターから空のコンピューターに青いアイテムを移動すると、ハードスコアが上がるはずです。以下のように、トラップされたスコア実装により、移動に失敗してしまいます。

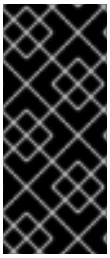


Solver は、最終的にこのトラップから外れますが、多くの労力を要します (特に、負荷の多いコンピューターのプロセスが多い場合など)。ペナルティーがないため、トラップから外すために、実際には負荷が多いコンピューターにプロセスを移動する可能性があります。

注記

スコアトラップを回避しても、スコア関数が局所最適条件を回避するほどスマートになるわけではありません。最適化アルゴリズムに、局所最適条件を処理させるようにしてください。

スコアトラップを回避すると、スコア制約ごとに、上下に変動しないスコア関数を回避することになります。



重要

実行不可能の程度を常に指定するようにしてください。ビジネスでは頻繁に、「ソリューションが実行できないものであれば、実行不可能なレベルは関係ない」と考えます。これは、ビジネスにとっては真実であってもスコア計算では異なるため、実行不可能なレベルを知ることで利点があります。実際には、ソフト制約では自然に実行されるので、ハード制約でも取り入れるだけなのです。

スコアトラップの使用方法は、複数存在します。

- スコア制約を改善して、スコアの重みを区別します。たとえば、CPU が足りない場合に **-1hard** のペナルティーを課すのではなく、足りていない CPU の数ごとに **-1hard** のペナルティーを課します。
- ビジネスの観点から、スコア制約を変更できない場合に、区別ができるスコア制約を使用して、レベルが低いスコアレベルを追加します。たとえば、CPU が足りない場合に **-1hard** を課し、さらに、足りない CPU の数だけ **-1subsoft** を課します。ビジネスでは、サブソフトスコアレベルが無視されます。
- 粒度の粗い Move を追加して、既存の粒度の細かい Move をまとめて選択します。粒度の粗い Move が、複数の Move を効率よく実行して、1 つの Move でスコアトラップから抜け出します。たとえば、同じコンテナーから別のコンテナーに複数アイテムを移動する場合などです。

5.4.9. stepLimit ベンチマーク

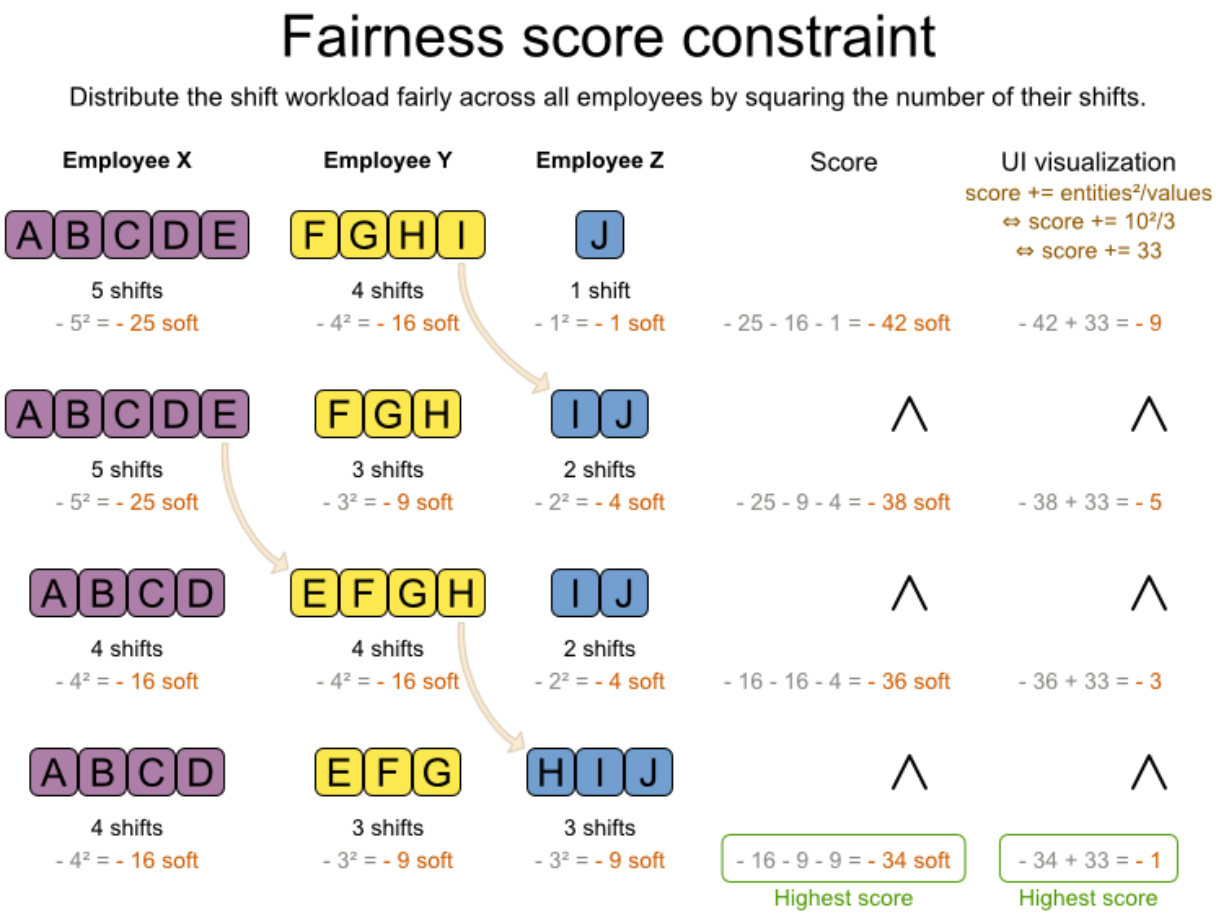
すべての制約に、同じパフォーマンスコストがかかるわけではありません。1 つのスコア制約により、スコア計算のパフォーマンス全体が落ちてしまう可能性があります。ベンチマーカー を使用して 1 分間実行し、スコア制約を 1 つだけ残してすべてコメントアウトした場合に、秒単位の平均計算数がどうなるかを確認します。

5.4.10. 公平なスコア制約

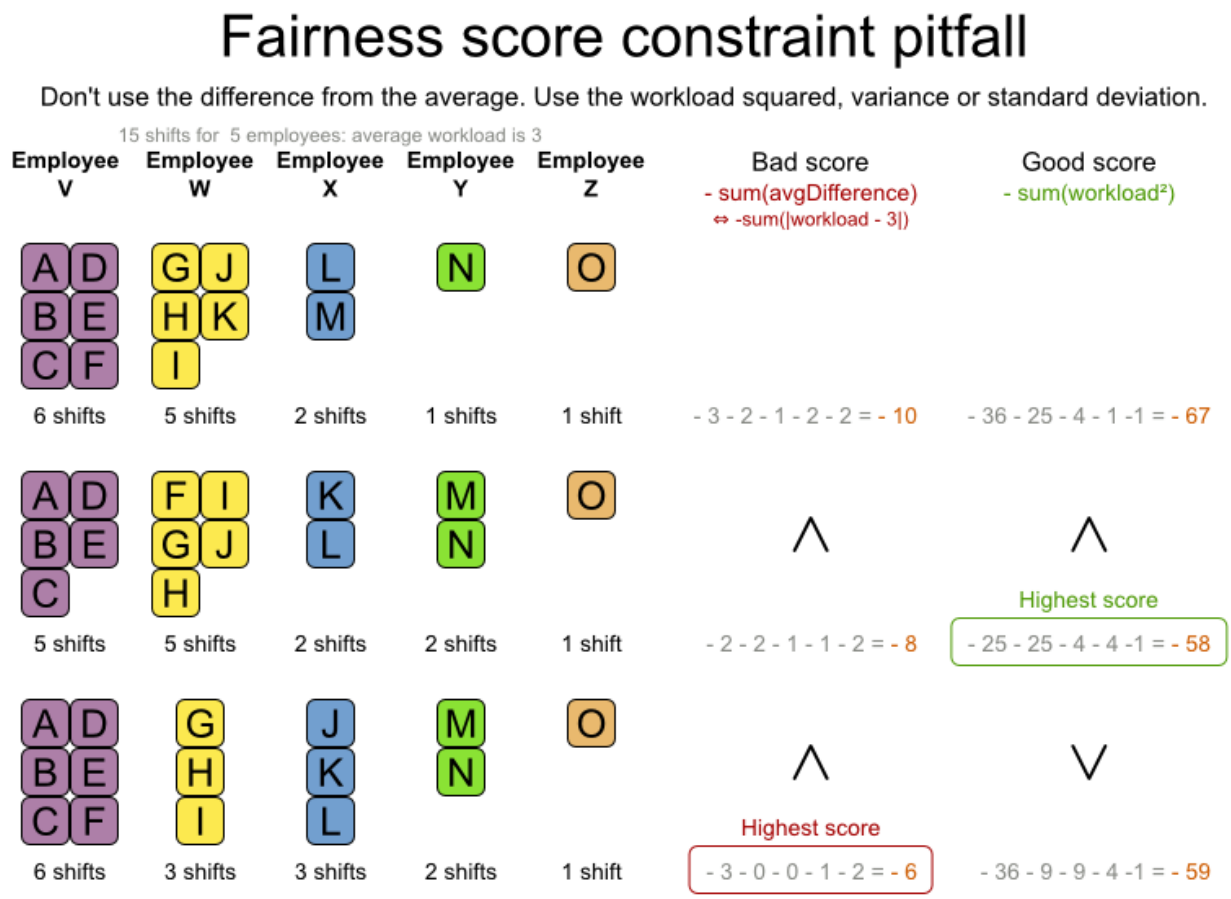
ユースケースによっては、ビジネス要件に公平なスケジュール (通常はソフトスコア制約として) が含まれる場合があります。以下に例を示します。

- 不満がでないように、従業員全体でワークロードを公平に分散します。
- アセット全体でワークロードを均等に分散して、信頼性を上げます。

(特に公平性を正式化する方法は複数あるため) このような制約を実装するのは困難なように見えますが、通常、ワークロードを 2 乗した実装が最も理想的に動作します。従業員/アセットごとに、ワークロード **w** をカウントし、スコアから w^2 を除算します。



上記のように、ワークロードを 2 乗した実装では、特定のソリューションから従業員 2 名を選択し、その 2 名の間で公平にワークロードを分散すると、新たなソリューションの全体スコアが改善されます。平均ワークロードからの差異だけを使用しないようにしてください。以下のように、不公平な結果になってしまう可能性があります。



注記

ワークロードを 2 乗する代わりに、分散 (平均からの差異を 2 乗) または 標準偏差 (分散からの平方根) を使用することもできます。平均は計画時に変更しないため、これはスコア比較には影響しません。実装作業が増え (平均値が必要)、明らかに処理が遅くなります (計算にかかる時間が増えます)。

ワークロードが均等に分散されている場合には、上の図のように紛らわしいスコア (**-34soft** (ほぼ均等に分散された最終解)) よりも、**0** スコアが好まれる場合が多くなります。これを null 化するには、スコアに、エンティティーの数を乗算した平均を追加するか、UI で分散または標準偏差を表示します。

5.5. スコアの説明: SOLVER 外でのスコア計算の使用

Web UI など、アプリケーションの他の部分で、スコアの計算が必要になる場合があります。これには、**Solver** の **ScoreDirectorFactory** を再利用して、その Web UI の別の **ScoreDirector** を構築します。

```
ScoreDirectorFactory scoreDirectorFactory = solver.getScoreDirectorFactory();
ScoreDirector guiScoreDirector = scoreDirectorFactory.buildScoreDirector();
```

次に、ソリューション の スコア を計算する必要がある場合に使用します。

```
guiScoreDirector.setWorkingSolution(solution);
Score score = guiScoreDirector.calculateScore();
```

どのエンティティーが スコア のどの部分に影響を与えているのかを GUI で説明するには、**ScoreDirector** から **ConstraintMatch** オブジェクトを取得します。

```
for (ConstraintMatchTotal constraintMatchTotal : guiScoreDirector.getConstraintMatchTotals()) {
    String constraintName = constraintMatchTotal.getConstraintName();
    Number weightTotal = constraintMatchTotal.getWeightTotalAsNumber();
    for (ConstraintMatch constraintMatch : constraintMatchTotal.getConstraintMatchSet()) {
        List<Object> justificationList = constraintMatch.getJustificationList();
        Number weight = constraintMatch.getWeightAsNumber();
        ...
    }
}
```

注記

Drools スコア計算 は、自動的に制約の一致をサポートしますが、Java インクリメント演算子によるスコア計算 には、追加のインターフェース (対象のセクションを参照) が必要になります。

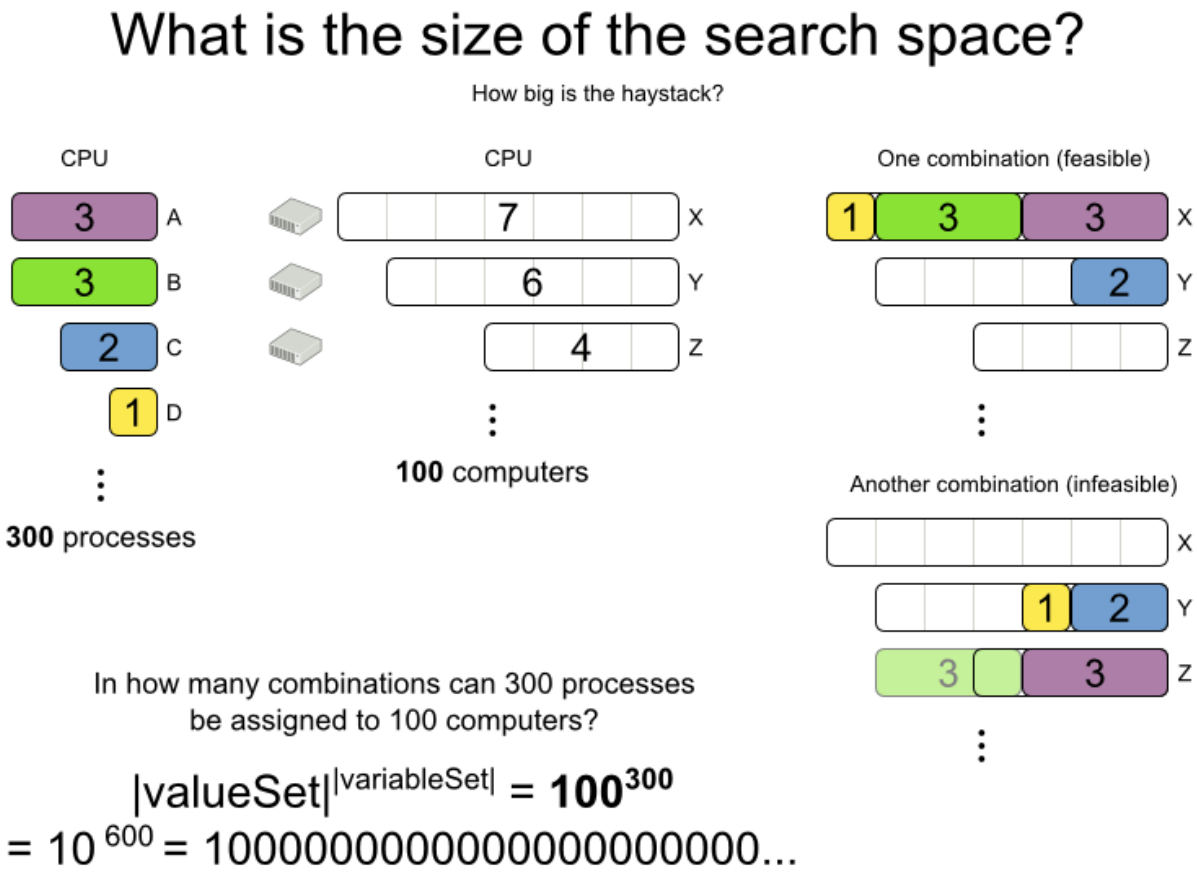
第6章 最適化アルゴリズム

6.1. 現実世界における探索空間のサイズ

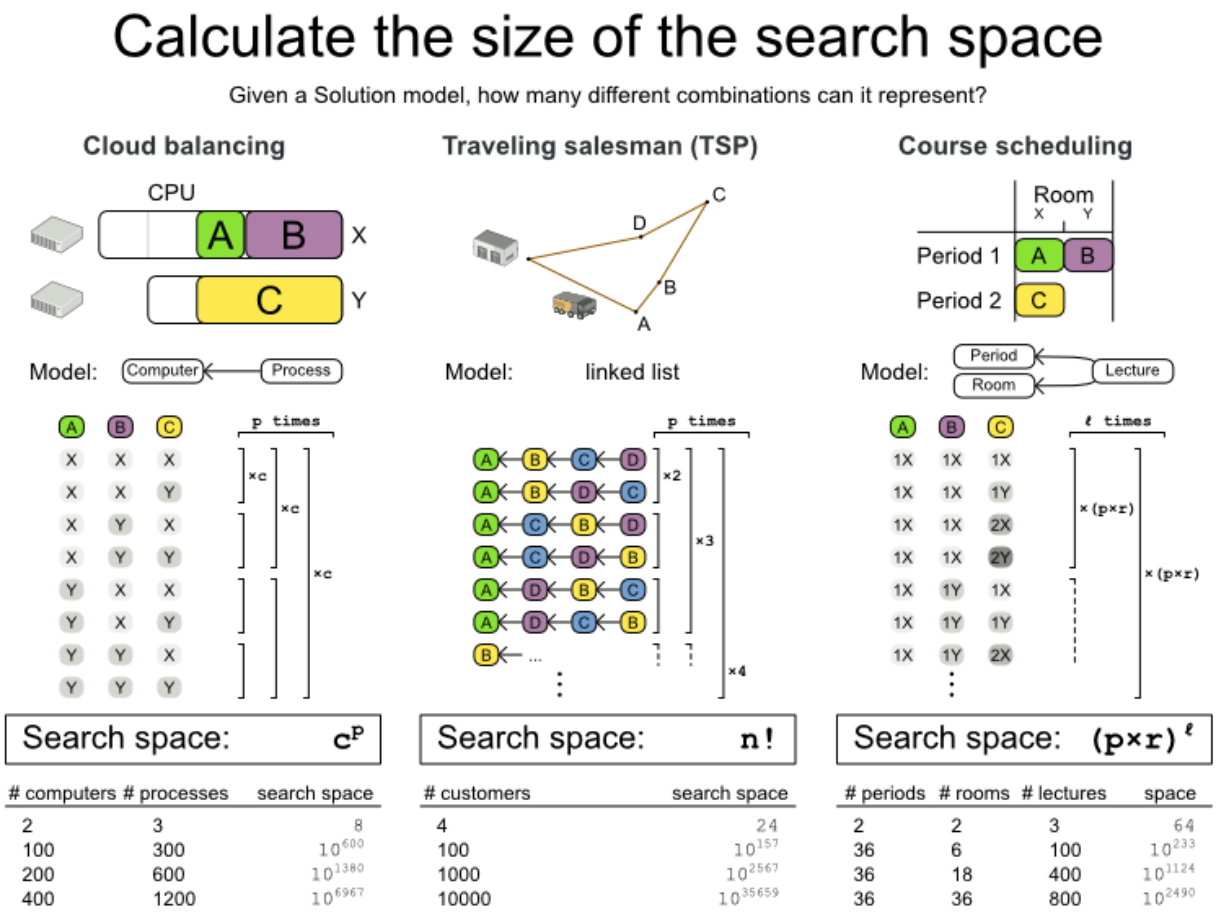
計画問題に対する可能解の数には圧倒されます。たとえば、以下のようになります。

- クイーンが 4 個の場合、可能解が 256 個 (4^4)、最適解が 2 個になります。
- クイーンが 5 個の場合、可能解が 3125 個 (5^5)、最適解が 1 個になります。
- クイーンが 8 個の場合、可能解が 16777216 個 (8^8)、最適解が 92 個になります。
- クイーンが 64 個になると、可能解は 10^{115} (64^{64}) を超えます。
- 現実世界の計画問題では、可能解の数は驚異的な数字となりますが、最適解となると、1 個もしくは数個になります。

なお、既知の宇宙に存在する原子の数の最小値は 10^{80} です。計画問題が大きくなればなるほど、探索空間は急激に大きくなります。したがって、プランニングエンティティーまたは計画値を 1 つ追加すると、アルゴリズムの実行時間が非常に長くなります。



計算する可能解の数は、ドメインモデルの設定によって異なります。



注記

この探索空間のサイズ計算には、以下の原因により (モデルが表示した場合は) 実行不可解も含まれます。

- 最適解が実行できない可能性がある。
- 実際には計算式に組み込めないハード制約が多数ある。たとえば、「クラウドのバランス」の例で、計算式に CPU 容量制約を組み込んでみてください。

「コースの時間割」など、計算式に追加するハード制約が実際には可能だったとしても、その探索空間は膨大になるものもあります。

可能解をすべて確認するアルゴリズム (分枝限定などの枝刈りでさえも) は、現実世界の計画問題を 1 つ実行するのに数億年かかる可能性があります。ここで必要なのは、限られた時間の中で自由に使える最適解を見つけることです。通常は、現実世界の制限を考えると、(International Timetabling Competition などの) 計画コンペティションでは、局所探索の各方法 (タブー探索、焼きなまし、レイトアクセプタンス など) が、能力を最大限に発揮します。

6.2. PLANNER では最適解が見つけられるのか?

ビジネスでは最適解が必要ですが、必須条件は他にもあります。

- スケールアウト: 本番のデータセットがクラッシュせず、良い結果が得られること。
- 正しい問題を最適化: 実際のビジネスニーズに制約が一致していること。

- 利用可能時間: 実行できなくなる前に解 (ソリューション) を見つけること。
- 信頼性: すべてのデータセットで、少なくとも (人が行う計画以上の) 適切な結果が得られること。

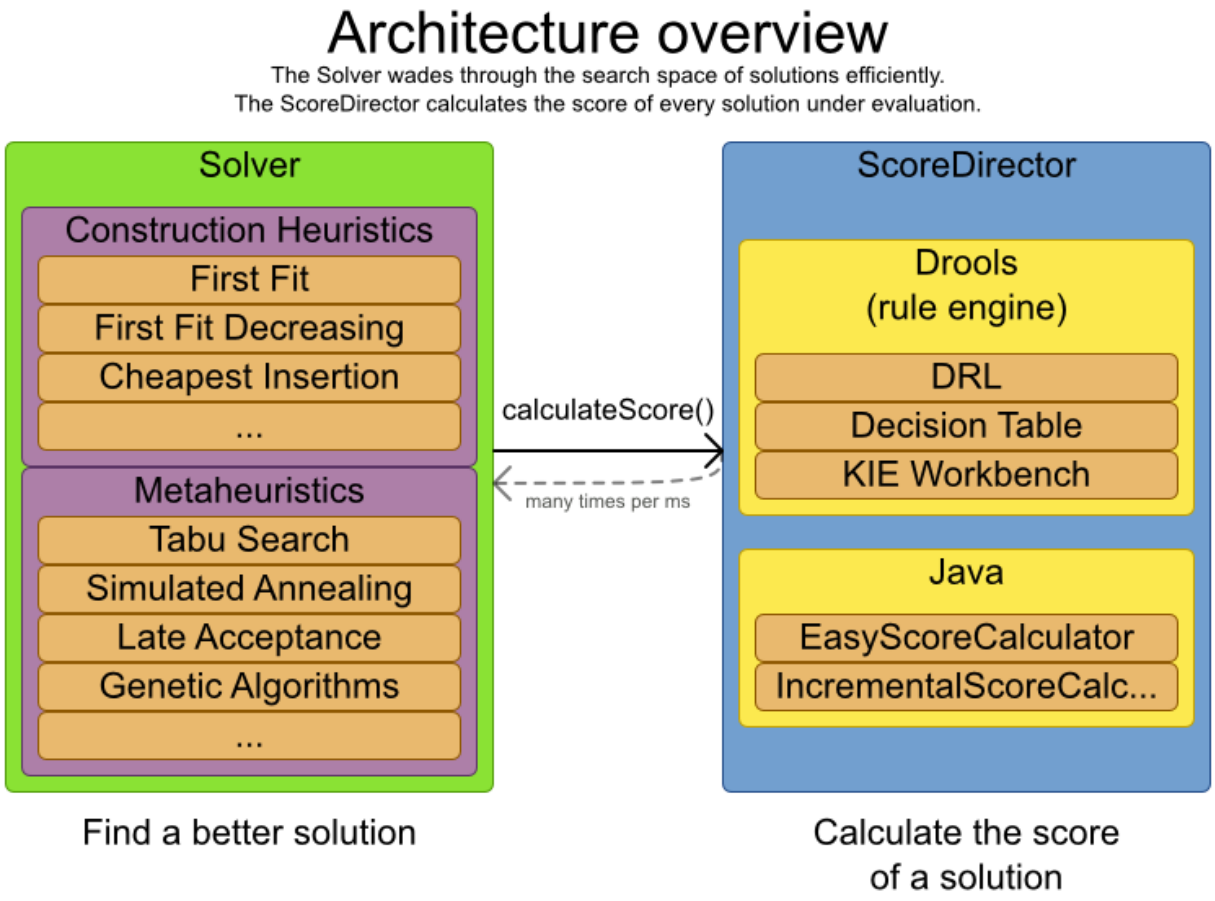
Given these requirements, and despite the promises of some salesmen, it's usually impossible for anyone or anything to find the optimal solution. Therefore, Planner focuses on finding the best solution in available time. In "realistic, independent competitions", it often comes out as the best **reusable** software.

NP 完全問題の性質は、最大の関心事をスケールします。データセットが小さいときの結果の品質が、データセットが大きくなったときの結果の品質を保証するものではありません。スケーリング問題は、のちにハードウェアを購入しても軽減されるものではありません。本番サイズのデータセットをできるだけ早めにテストするようにしてください。品質は、データセットが小さい時に評価しないでください (本番でも同規模のデータセットを使用する場合は除きます)。代わりに、本番サイズのデータセットを解決し、アルゴリズムを変えて長時間実行したときの結果と比較します (可能な場合は手動での計画の結果も比較します)。

6.3. アーキテクチャーの概要

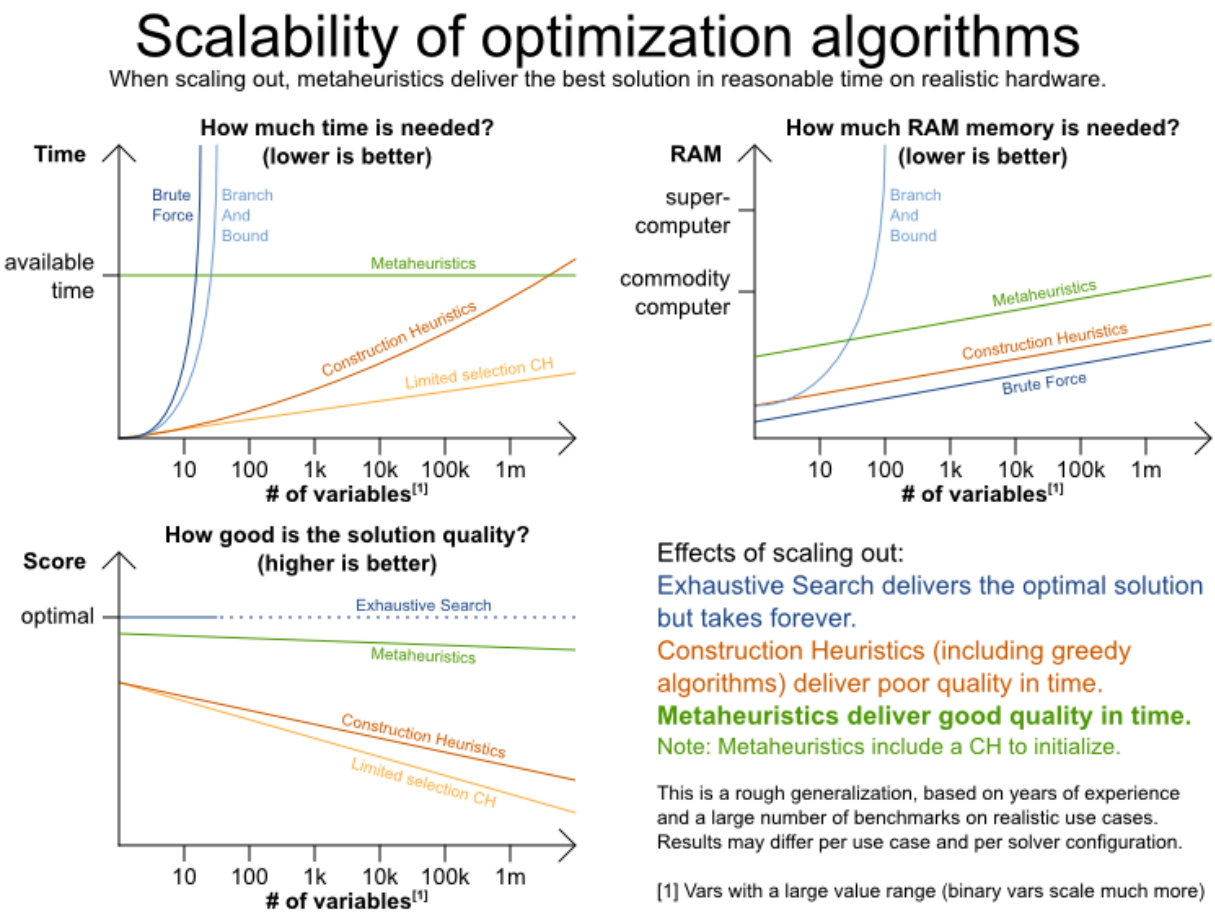
Planner は、最適化アルゴリズム (メタヒューリスティクなど) と、ルールエンジン (Drools Expert など) によるスコア計算を比較する、初めてのフレームワークです。この組み合わせは、以下の理由により、非常に効率的です。

- Drools Expert などのルールエンジンは、計画問題のソリューションのスコアを計算するのに優れています。「教師は、1 日 7 時間以上指導はしてはいけない」などのソフトまたはハードの制約を追加して簡単にスケールすることができます。また、差分ベースのスコア計算が、コードを追加しなくても可能です。ただし、これは、実際に新しいソリューションを見つけるのには適していません。
- 最適化アルゴリズムは、計画問題において、すべてのソリューションをしらみつぶしに評価しなくても、新たに改善したソリューションを見つけるのに適しています。ただし、それにはソリューションのスコアが必要で、そのスコアを効率的に計算することには対応していません。



6.4. 最適化アルゴリズムの概要

Planner は、最適化アルゴリズムの 3 つのファミリー (しらみつぶし探索、構築ヒューリスティック、およびメタヒューリスティク) に対応しています。選択肢としては、(構築ヒューリスティックと組み合わせて初期化する) メタヒューリスティックが推奨されます。



各ファミリーのアルゴリズムには、複数の最適化アルゴリズムがあります。

表6.1 最適化アルゴリズムの概要

アルゴリズム	拡張可能か?	最適か?	使いやすいか?	調整可能か?	CH が必要か?
Exhaustive Search (しらみつぶし探索) (ES)					
Brute Force (力まかせ)	☆☆☆☆☆	★★★★★	★★★★★	☆☆☆☆☆	なし
Branch And Bound (分枝限定法)	☆☆☆☆☆	★★★★★	★★★★☆	★★☆☆☆	なし
Construction heuristics (構築ヒューリスティック) (CH)					
First Fit (FF)	★★★★★	★☆☆☆☆	★★★★★	★☆☆☆☆	なし
First Fit Decreasing (FFD)	★★★★★	★★☆☆☆	★★★★☆	★★☆☆☆	なし

アルゴリズム	拡張可能か?	最適か?	使いやすいか?	調整可能か?	CH が必要か?
Weakest Fit (WF)	★★★★★	★★☆☆☆	★★★★☆	★★☆☆☆	なし
Weakest Fit Decreasing (WFD)	★★★★★	★★☆☆☆	★★★★☆	★★☆☆☆	なし
Strongest Fit (SF)	★★★★★	★★☆☆☆	★★★★☆	★★☆☆☆	なし
Strongest Fit Decreasing (SFD)	★★★★★	★★☆☆☆	★★★★☆	★★☆☆☆	なし
Cheapest Insertion (CI)	★★★☆☆	★★☆☆☆	★★★★★	★★☆☆☆	なし
Regret Insertion (RI)	★★★☆☆	★★☆☆☆	★★★★★	★★☆☆☆	なし
Metaheuristics (メタヒューリスティク) (MH)					
Local Search (局所探索法)					
Hill Climbing (山登り法)	★★★★★	★★☆☆☆	★★★★☆	★★★☆☆	あり
Tabu Search (タブー探索)	★★★★★	★★★★☆	★★★☆☆	★★★★★	あり
Simulated Annealing (焼きなまし法)	★★★★★	★★★★☆	★★☆☆☆	★★★★★	あり
Late Acceptance (レイトアクセプタンス)	★★★★★	★★★★☆	★★★☆☆	★★★★★	あり
Step Counting Hill Climbing (SCHC)	★★★★★	★★★★☆	★★★☆☆	★★★★★	あり
Evolutionary Algorithms (進化アルゴリズム)					

アルゴリズム	拡張可能か?	最適か?	使いやすいか?	調整可能か?	CH が必要か?
Evolutionary Strategies (進化ストラテジー)	★★★★☆	★★★☆☆	★★☆☆☆	★★★★★	あり
Genetic Algorithms (遺伝的アルゴリズム)	★★★★☆	★★★☆☆	★★☆☆☆	★★★★★	あり

メタヒューリスティクの詳細は、無料の書籍『Essentials of Metaheuristics』または『Clever Algorithms』を参照してください。

6.5. どの最適化アルゴリズムを使用すべきか?

ユースケースによって、一番適した 最適化アルゴリズム設定は異なります。ただし、以下の一般的な方法に比べると非常に優れており、おそらくこれまで使用していたものよりもはるかに優れています。

設定と最適化コードには全く、もしくはほとんど関与しない簡単な設定から始めます。

1. First Fit (FF)

次に、計画エンティティの難易度の比較を実装し、以下のように変更します。

1. First Fit Decreasing (FFD)

次に、レイトアクセプタンスを追加します。

1. First Fit Decreasing (FFD)
2. Late Acceptance (レイトアクセプタンス)。通常、レイトアクセプタンスを 400 にすると適切に機能します。

ここまでは簡単です。これ以上時間をかけても、時間に対して得られる結果は少なくなります。また、この時点で得られた結果は、十分優れています。

ただし、かかった時間に対して得られるものは少なくなりますが、さらに優れた結果を得ることはできます。ベンチマーカー を使用して、異なる設定 (タブー探索、焼きなまし法、レイトアクセプタンス など) をいくつか試します。

1. First Fit Decreasing (FFD)
2. Tabu Search (タブー探索)。エンティティーのタブーの大きさを 7 にすると、通常は適切に機能します。

ベンチマーカー を使用して、このサイズパラメーターの値を改善します。

時間をかける価値がある場合は、さらに検証を行います。複数のアルゴリズムを組み合わせてみてください。

1. First Fit Decreasing (FFD)

2. Late Acceptance (レイトアクセプタンス) (時間は比較的長くなります)

3. Tabu Search (タブー探索) (時間は比較的短くなります)

6.6. 調整またはパラメーターのデフォルト値

多くの最適化アルゴリズムには、結果とスケーラビリティーに影響するパラメーターがあります。Planner は、例外による設定を提供するため、すべての最適化アルゴリズムにはデフォルトのパラメーター値があります。これは、JVM のガベージコレクションパラメーターに非常に良く似ているため、多くの場合はこれを調整する必要はありませんが、上級者は調整してください。

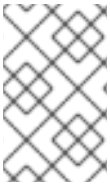
デフォルトのパラメーターは、多くの場合 (特にプロトタイプに対して) 十分優れていますが、開発時間に余裕がある場合は、さらに良い結果を得るために、ベンチマーカー を使用して調整する価値は十分あります。それぞれの最適化アルゴリズムに関するドキュメントでは、調整を行う高度な設定も示します。

警告

マイナーバージョン間におけるパラメーターのデフォルト値は、多くのユーザーを対象にした機能改善を目的に、変更されます (すべての場合に改善するとは限りません)。良くも悪くも、この変更の影響を受けないようにするには、常に高度な設定を使用するようにしてください。ただし、これは推奨はされません。

6.7. SOLVER のフェーズ

Solver は、複数の最適化アルゴリズムを順番に使用できます。最適化アルゴリズムは、それぞれ Solver の フェーズ として示されます。複数の フェーズ のソリューションを同時に導出することはありません。



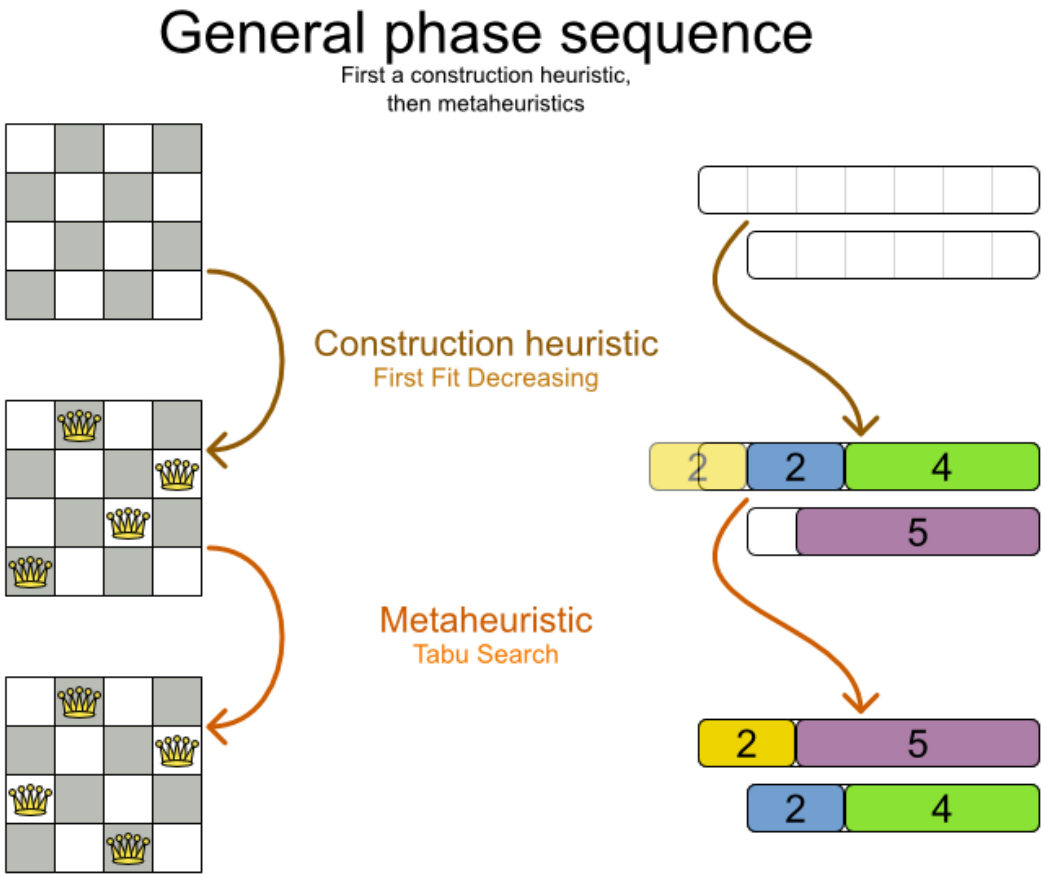
注記

一部の フェーズ実装では、複数の最適化アルゴリズムのテクニックを組み合わせることができますが、フェーズ の数としては 1 つとなります。たとえば、局所探索のフェーズでは、エンティティーのタブーを使用した焼きなまし法を行うことができます。

以下の設定では、3 つのフェーズが順番に実行します。

```
<solver>
  ...
  <constructionHeuristic>
    ... <!-- First phase: First Fit Decreasing -->
  </constructionHeuristic>
  <localSearch>
    ... <!-- Second phase: Late Acceptance -->
  </localSearch>
  <localSearch>
    ... <!-- Third phase: Tabu Search -->
  </localSearch>
</solver>
```

Solver のフェーズは、Solver の設定で指定した順番で実行します。最初の フェーズ が終了すると次の フェーズ が開始し、それが終了すると次が開始します。最後の フェーズ が終了すると、**Solver** が終了します。通常、**Solver** は最初に構築ヒューリスティックを実行してから、メタヒューリスティックスを 1 つまたは複数実行します。



フェーズが設定されていない場合、Planner は、デフォルトで、構築ヒューリスティックフェーズ、局所探索法フェーズの順に行います。

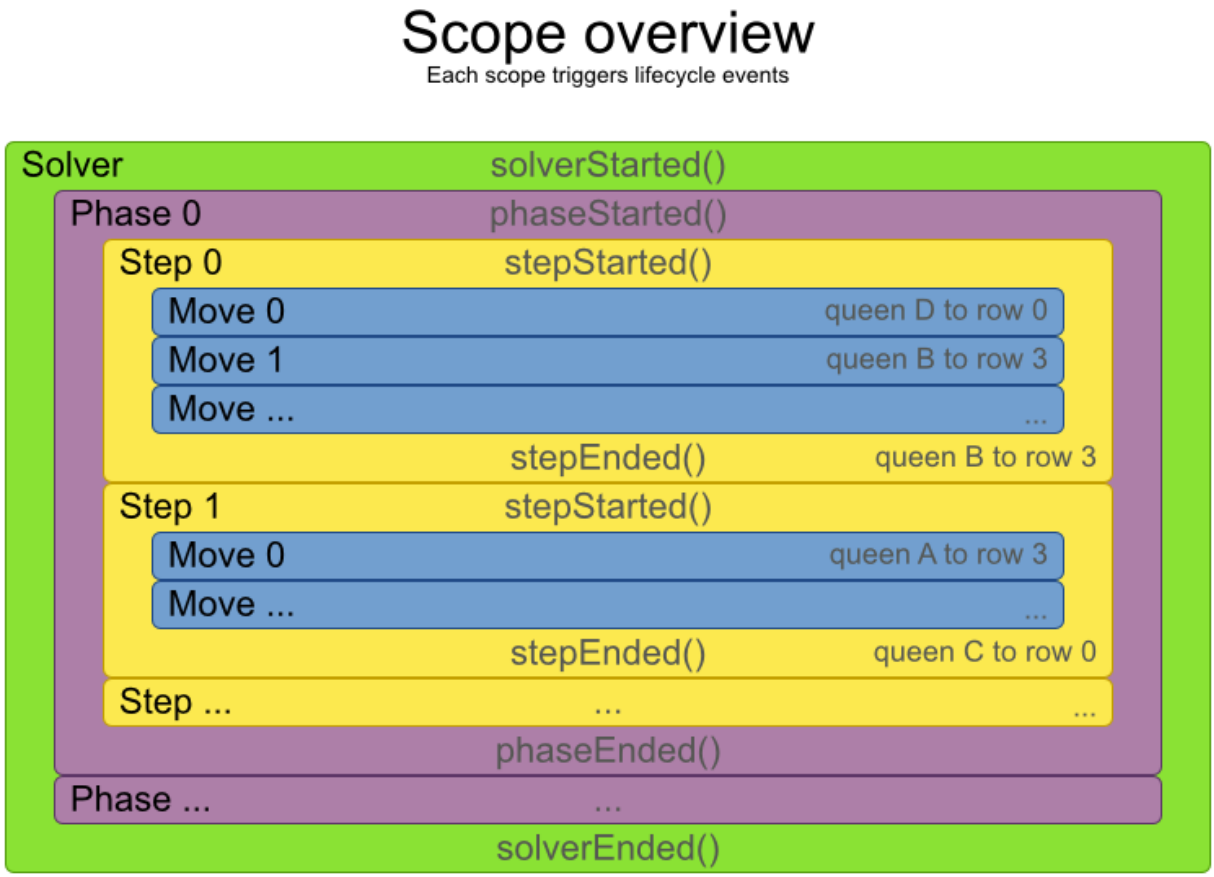
一部のフェーズ (特に構築ヒューリスティック) は自動的に終了します。自動的に終了しないフェーズ (特にメタヒューリスティック) は、フェーズが終了するように設定している場合に限り終了します。

```
<solver>
  ...
  <termination><!-- Solver termination -->
    <secondsSpentLimit>90</secondsSpentLimit>
  </termination>
  <localSearch>
    <termination><!-- Phase termination -->
      <secondsSpentLimit>60</secondsSpentLimit><!-- Give the next phase a chance to run too,
before the Solver terminates -->
    </termination>
    ...
  </localSearch>
  <localSearch>
    ...
  </localSearch>
</solver>
```

(最後の フェーズ が完了する前に) **Solver** が終了すると、現在のフェーズが終了し、後続のフェーズは実行されません。

6.8. スコープの概要

Solver は、フェーズを繰り返し実行します。各フェーズは、通常、Move を振り返し実行します。したがって、4 つのスコープ (Solver、フェーズ、ステップ、Move) がネスト状態となります。

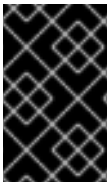


ログを設定して、各スコープのログメッセージを表示します。

6.9. 終了

すべてのフェーズが自動的に終了するとは限らず、また終了するまで待たずに終わらせる場合もあります。**Solver** は、事前設定により同期的に、または別のスレッドから非同期に終了できます。

特にメタヒューリスティックフェーズでは、問題解決を終了するタイミングを設定する必要があります。たとえば、時間が終了した時、完全スコアに到達した時、もしくはそのソリューションが使用される直前などです。使用できない唯一の設定は、最適解を見つけることです (ただし、最適解が分かっている場合は可能です)。なぜなら、通常、メタヒューリスティックアルゴリズムが、いつ最適解を見つけるか分からないからです。現実世界では、これは大きな問題にはなりません。なぜなら、最適解を見つけるのに何年もかかることがあり、それよりも早く終了したくなるためです。これが問題となるのは、利用可能な期間に最適解を見つけられる場合です。



重要

終了が設定されておらず、メタヒューリスティックアルゴリズムが使用されている場合は、別のスレッドから terminateEarly() が呼び出されるまで、**Solver** は永久に継続します。これは、特に リアルタイム計画 で一般的です。

同期終了については、**Solver** または フェーズ の 終了 で、停止するタイミングを設定します。自分で作成した 終了 を実装することもできますが、組み込みの実装でも大概のニーズを満たします。すべての 終了 で、(一部の最適化アルゴリズムに必要な) 時間勾配 を計算できます。これは、**Solver** または フェーズ で問題解決にかかると見積もられている時間に対する、実際にこれまでかかった時間の割合です。

6.9.1. TimeMillisSpentTermination

指定した時間が経過したら終了します。

```
<termination>
  <millisecondsSpentLimit>500</millisecondsSpentLimit>
</termination>
```

```
<termination>
  <secondsSpentLimit>10</secondsSpentLimit>
</termination>
```

```
<termination>
  <minutesSpentLimit>5</minutesSpentLimit>
</termination>
```

```
<termination>
  <hoursSpentLimit>1</hoursSpentLimit>
</termination>
```

```
<termination>
  <daysSpentLimit>2</daysSpentLimit>
</termination>
```

時間の種類は組み合わせて使用できます。たとえば、設定した時間が 150 分の場合は、以下のようにその値を直接設定します。

```
<termination>
  <minutesSpentLimit>150</minutesSpentLimit>
</termination>
```

もしくは、合計で 150 分になるように組み合わせて指定できます。

```
<termination>
  <hoursSpentLimit>2</hoursSpentLimit>
  <minutesSpentLimit>30</minutesSpentLimit>
</termination>
```

注記

この 終了 を使用すると、実行ごとに利用可能な CPU 時間が異なることが多いので、(**environmentMode REPRODUCIBLE** を使用していても) 完全再現性が犠牲になる可能性が非常に高くなります。

- 利用可能な CPU 時間は、実行するステップの数に影響し、少し増減する可能性があります。
- 終了 では、時間勾配の値がわずかに異なる可能性があります。これにより、まったく異なるパスに時間勾配アルゴリズム (焼きなまし法) が送られます。

6.9.2. UnimprovedTimeMillisSpentTermination

指定した時間内に最高スコアが変更しないと終了します。

```
<localSearch>
  <termination>
    <unimprovedMillisecondsSpentLimit>500</unimprovedMillisecondsSpentLimit>
  </termination>
</localSearch>
```

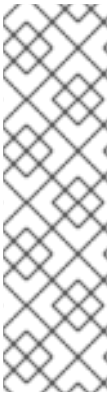
```
<localSearch>
  <termination>
    <unimprovedSecondsSpentLimit>10</unimprovedSecondsSpentLimit>
  </termination>
</localSearch>
```

```
<localSearch>
  <termination>
    <unimprovedMinutesSpentLimit>5</unimprovedMinutesSpentLimit>
  </termination>
</localSearch>
```

```
<localSearch>
  <termination>
    <unimprovedHoursSpentLimit>1</unimprovedHoursSpentLimit>
  </termination>
</localSearch>
```

```
<localSearch>
  <termination>
    <unimprovedDaysSpentLimit>1</unimprovedDaysSpentLimit>
  </termination>
</localSearch>
```

構築ヒューリスティックでは、最適解が更新されるのは最後だけなので、この終了は構築ヒューリスティックには適用されません。したがって、**Solver** そのものではなく、(**<localSearch>** などの) 特定の フェーズに設定することが推奨されます。



注記

この 終了 を使用すると、実行ごとに利用可能な CPU 時間が異なることが多いので、(**environmentMode REPRODUCIBLE** を使用していても) 完全再現性が犠牲になる可能性が非常に高くなります。

- 利用可能な CPU 時間は、実行するステップの数に影響し、少し増減する可能性があります。
- 終了 では、時間勾配の値がわずかに異なる可能性があります。これにより、まったく異なるパスに時間勾配アルゴリズム (焼きなまし法) が送られます。

6.9.3. BestScoreTermination

特定のスコアに到達すると終了します。この 終了 は、(たとえばクイーンが 4 個の場合に) 完全スコアが分かっている場合に限り使用します (これには SimpleScore が使用されます)。

```
<termination>
  <bestScoreLimit>0</bestScoreLimit>
</termination>
```

計画問題に HardSoftScore が指定されている場合は、以下のようになります。

```
<termination>
  <bestScoreLimit>0hard/-5000soft</bestScoreLimit>
</termination>
```

計画問題に、BendableScore が指定されており、ハードレベルが 3 で、ソフトレベルが 1 の場合は、以下のようになります。

```
<termination>
  <bestScoreLimit>0/0/0/-5000</bestScoreLimit>
</termination>
```

この終了には **bestScoreLimit** (**0hard/-2147483648soft** など) が必要になるため、実行可能解に到達したら終了するように設定する場合は、この 終了 は実用的ではありません。代わりに、次の終了を使用します。

6.9.4. BestScoreFeasibleTermination

特定のスコアが実行可能な場合は終了します。スコア 実装に **FeasibilityScore** が実装されている必要があります。

```
<termination>
  <bestScoreFeasible>true</bestScoreFeasible>
</termination>
```

この 終了 は、通常は、別の終了と組み合わせます。

6.9.5. StepCountTermination

ステップの数に到達したら終了します。これは、ハードウェアパフォーマンスに依存せずに実行している場合に役に立ちます。

```
<localSearch>
  <termination>
    <stepCountLimit>100</stepCountLimit>
  </termination>
</localSearch>
```

この 終了 は、フェーズ (**<localSearch>** など) にのみ使用でき、**Solver** 自身には使用できません。

6.9.6. UnimprovedStepCountTermination

多数のステップを行っても最高スコアが変わらない場合に終了します。これは、ハードウェアパフォーマンスに依存せずに実行している場合に便利です。

```
<localSearch>
  <termination>
    <unimprovedStepCountLimit>100</unimprovedStepCountLimit>
  </termination>
</localSearch>
```

スコアがしばらく変わらず、おそらく今後もしばらく変わることがない場合は、継続しても有効ではありません。新たに最適解が見つかったら、(しばらく最適解が変わっていない場合でも) 次の数ステップで最適解が変わる傾向にあることが分かりました。

この 終了 は、フェーズ (**<localSearch>** など) にのみ使用でき、**Solver** 自身には使用できません。

6.9.7. CalculateCountTermination

スコア計算の数 (通常は Move の数とステップの数の合計) に到達したら終了します。これはベンチマークに使用すると役に立ちます。

```
<termination>
  <calculateCountLimit>100000</calculateCountLimit>
</termination>
```

EnvironmentMode に切り替えると、この終了が終わるタイミングに大きな影響を及ぼす可能性があります。

6.9.8. 複数の終了の組み合わせ

終了は組み合わせて使用できます。たとえば、以下の例では、ステップ数が **100** になるか、スコアが **0** がなったら終了します。

```
<termination>
  <terminationCompositionStyle>OR</terminationCompositionStyle>
  <stepCountLimit>100</stepCountLimit>
  <bestScoreLimit>0</bestScoreLimit>
</termination>
```

また、AND も使用できます。以下の例では、実行可能解の数が **-100** になり、**5** ステップ中に値が改善しない場合は終了します。

```
<termination>
  <terminationCompositionStyle>AND</terminationCompositionStyle>
```

```
    <unimprovedStepCountLimit>5</unimprovedStepCountLimit>
    <bestScoreLimit>-100</bestScoreLimit>
  </termination>
```

この例では、実行可能解を見つけただけでは終了せず、その解における明確な変化が完了した場合に終了します。

6.9.9. 別のスレッドからの非同期終了

ユーザー操作、またはサーバーの再起動で、別のスレッドから Solver を早期終了させる場合もあります。終了の有無やタイミングを予測することができないため、終了 を使って設定することはできません。したがって、**Solver** インターフェースには、このようなスレッドセーフメソッドが 2 つあります。

```
public interface Solver<S extends Solution> {

    // ...

    boolean terminateEarly();
    boolean isTerminateEarly();

}
```

terminateEarly() メソッドを別のスレッドから呼び出すと、**Solver** は、終了が可能になり次第終了し、(元の **Solver** スレッドで) **solve(Solution)** メソッドが返ります。

注記

Solver スレッド (**Solver.solve(Solution)** を呼び出すスレッド) を中断することは、**terminateEarly()** を呼び出すのと同じ影響がありますが、スレッドが中断状態のままになる点だけが異なります。これにより、**ExecutorService** (スレッドプールなど) がシャットダウンするときに正常なシャットダウンが行われます。なぜなら、これにより、プール内でアクティブなスレッドだけをすべて中断するためです。

6.10. SOLVEREVENTLISTENER

最適解が新たに見つかるたびに、**Solver** は 、Solver のスレッドで **BestSolutionChangedEvent** を発生させます。

このようなイベントをリッスンするには、**SolverEventListener** を **Solver** に追加します。

```
public interface Solver<S extends Solution> {

    // ...

    void addEventListener(SolverEventListener<S> eventListener);
    void removeEventListener(SolverEventListener<S> eventListener);

}
```

BestSolutionChangedEvent の **newBestSolution** は初期化されていないか実行可能ではない場合があります。このような状況を検出するには、**BestSolutionChangedEvent** でこのメソッドを使用します。

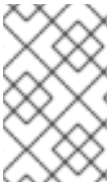
```
solver.addEventListener(new SolverEventListener<CloudBalance>() {
    public void bestSolutionChanged(BestSolutionChangedEvent<CloudBalance> event) {
        // Ignore invalid solutions
        if (event.isNewBestSolutionInitialized()
                && event.getNewBestSolution().getScore().isFeasible()) {
            ...
        }
    }
});
```

警告

bestSolutionChanged() メソッドは、問題解決が遅くならないように速やかに解 (ソリューション) を返し、**Solver.solve()** の一部として、Solver のスレッドで呼び出されます。

6.11. カスタムの SOLVER フェーズ

フェーズ全体を実装せずにカスタムの構築ヒューリスティックを実装する場合など、フェーズ間、または最初のフェーズの前に、カスタムの最適化アルゴリズムを実行して、ソリューション を初期化したり、簡単に達成できる目標を定めてより良いスコアをすぐに得たりするだけでなく、スコア計算を再利用する場合があります。



注記

大抵の場合、Solver フェーズをカスタマイズする価値はありません。対応している 構築ヒューリスティック は設定で変えられ (調整には ベンチマーカー を使用)、終了 は、一部初期化されたソリューションも認識、対応します。

CustomPhaseCommand インターフェースは以下のようになります。

```
public interface CustomPhaseCommand {

    void applyCustomProperties(Map<String, String> customPropertyMap);

    void changeWorkingSolution(ScoreDirector scoreDirector);

}
```

たとえば、**AbstractCustomPhaseCommand** を拡張し、**changeWorkingSolution()** メソッドを実装します。

```
public class ToOriginalMachineSolutionInitializer extends AbstractCustomPhaseCommand {

    public void changeWorkingSolution(ScoreDirector scoreDirector) {
        MachineReassignment machineReassignment = (MachineReassignment)
scoreDirector.getWorkingSolution();
        for (MrProcessAssignment processAssignment :
machineReassignment.getProcessAssignmentList()) {
```

```
            scoreDirector.beforeVariableChanged(processAssignment, "machine");
            processAssignment.setMachine(processAssignment.getOriginalMachine());
            scoreDirector.afterVariableChanged(processAssignment, "machine");
            scoreDirector.triggerVariableListeners();
        }
    }

}
```

警告

CustomPhaseCommand でプランニングエンティティーを変更した場合は、**ScoreDirector** に通知する必要があります。

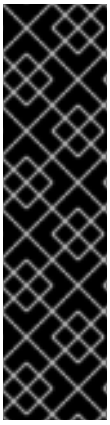
警告

CustomPhaseCommand における問題ファクトを変更しないでください。変更すると、以前のスコアやソリューションが別の問題のものになり、**Solver** が壊れます。これを行うには、代わりに、反復計画 に記載されている ProblemFactChange を使用してください。

CustomPhaseCommand を以下のように設定します。

```
<solver>
  ...
  <customPhase>

<customPhaseCommandClass>org.optaplanner.examples.machinereassignment.solver.solution.initializer.ToOriginalMachineSolutionInitializer</customPhaseCommandClass>
  </customPhase>
  ... <!-- Other phases -->
</solver>
```

複数の **customPhaseCommandClass** インスタンスが順番に実行するように設定します。



重要

CustomPhaseCommand を変更してもスコアが良くならない場合は、最適解が変更しません (したがって、実際には、次の フェーズ または **CustomPhaseCommand** で何も変更しません)。このような変更を強制するには、**forceUpdateBestSolution** を使用します。

```
<customPhase>
  <customPhaseCommandClass>...MyUninitializer</customPhaseCommandClass>
  <forceUpdateBestSolution>true</forceUpdateBestSolution>
</customPhase>
```

注記

CustomPhaseCommand が実行中の場合でも、**Solver** または フェーズ を終了する場合は、**CustomPhaseCommand** が終わるまで待ちます。ただし、これには時間がかかります。組み込み Solver フェーズでは、この問題の影響を受けません。

Solver 設定で **CustomPhaseCommand** の値を動的に設定するには (ベンチマーカー でこのパラメーターを調整可能)、**customProperties** 要素を使用します。

```
<customPhase>
  <customProperties>
    <mySelectionSize>5</mySelectionSize>
  </customProperties>
</customPhase>
```

次に、**applyCustomProperties()** メソッドを上書きして、**Solver** を構築する時に解析して適用します。

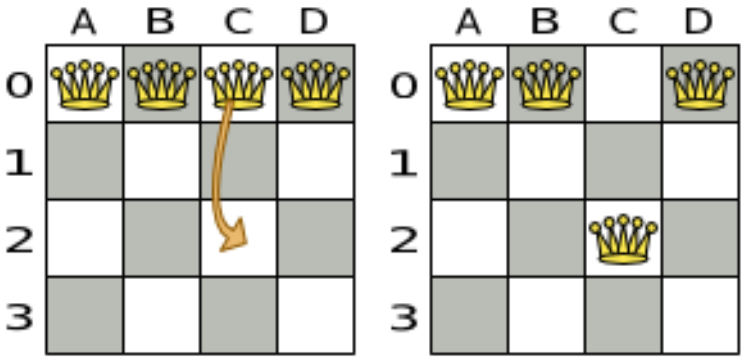
```
public class MySolutionInitializer extends AbstractCustomPhaseCommand {

    private int mySelectionSize;

    public void applyCustomProperties(Map<String, String> customPropertyMap) {
        String mySelectionSizeString = customPropertyMap.get("mySelectionSize");
        if (mySelectionSizeString == null) {
            throw new IllegalArgumentException("A customProperty (mySelectionSize) is missing from
the solver configuration.");
        }
        solverFactory = SolverFactory.createFromXmlResource(partitionSolverConfigResource);
        if (customPropertyMap.size() != 1) {
            throw new IllegalArgumentException("The customPropertyMap's size (" +
customPropertyMap.size() + ") is not 1.");
        }
        mySelectionSize = Integer.parseInt(mySelectionSizeString);
    }

    ...
}
```

第7章 MOVE と近傍選択

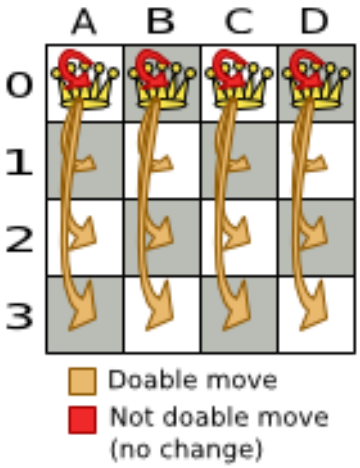
7.1. MOVE および近傍の概要

7.1.1. Move について

Move は、解 A から解 B への変更 (または一連の変更) です。たとえば、以下の Move では、クイーン **C** が行 **0** から行 **2** に変更します。

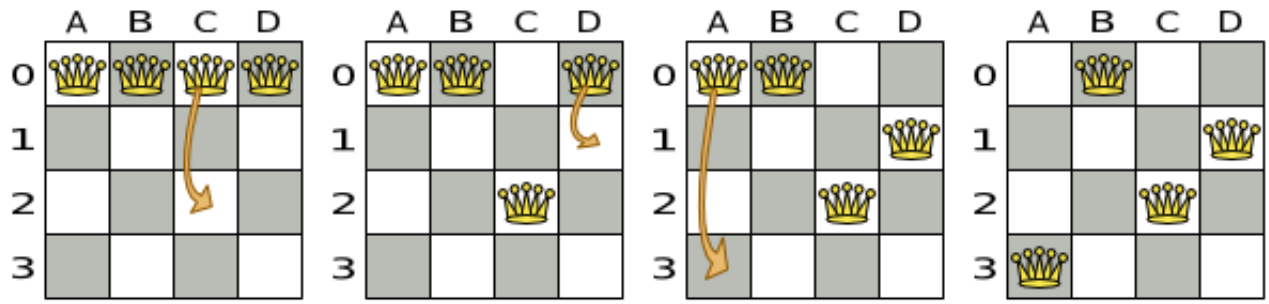


新しい解は、1 つの **Move** で到達できるため、元の解の近傍と呼ばれます。1 つの Move で複数のクイーンを変更できたしても、解の近傍値は、常にすべての可能解の、非常に小さいサブセットとなります。たとえば、その元の解から見ると、可能なすべての **changeMove** です。



4 つの **changeMove** は影響を及ぼさず、実行可能ではないため、この 4 つを無視した場合の Move の数は **n * (n - 1) = 12** です。これは、可能解の数 (n^n **= 256**) に比べてはるかに少なくなります。問題の増え方と比較した、可能な Move の数は、可能解の数に比べてはるかに少なくなります。

にもかかわらず、**changeMove** が 4 以下の場合は、どの解にも到達できます。たとえば、3 個の **changeMove** で到達できる解は大変異なります。



注記

changeMove 以外にも、Move には多くの種類があります。多くの Move はデフォルトで追加されていますが、カスタムの Move も実装できます。

Move は、複数のエンティティーに影響を与え、エンティティーの作成または削除も可能ですが、問題ファクトは変更しないでください。

すべての最適化アルゴリズムは、**Move** を使用して、ある解から近傍解に移行します。したがって、最適解アルゴリズムは、**Move** の選択を迫られます。効果的に Move を作成および反復する技術と、最初に評価するランダムの Move サブセットの中から、最も見込みのあるものを見つける技術になります。

7.1.2. MoveSelector について

MoveSelector の主な機能は、必要に応じて **Iterator<Move>** を作成することです。最適化アルゴリズムは、この Move のサブセットの繰り返しとなります。

以下は、最適化アルゴリズムの局所探索法に **changeMoveSelector** を設定する方法になります。

```
<localSearch>
  <changeMoveSelector/>
  ...
</localSearch>
```

これは、何も設定しなくても機能し、**changeMoveSelector** の全プロパティーのデフォルトは実用的です (曖昧さによるフェイルファーストは除きます)。一方、設定は特定のユースケースに合わせてかなりカスタマイズできます。たとえば、意味のない Move を破棄するフィルターを設定することもできます。

7.1.3. エンティティ、値、およびその他の Move のサブセレクト

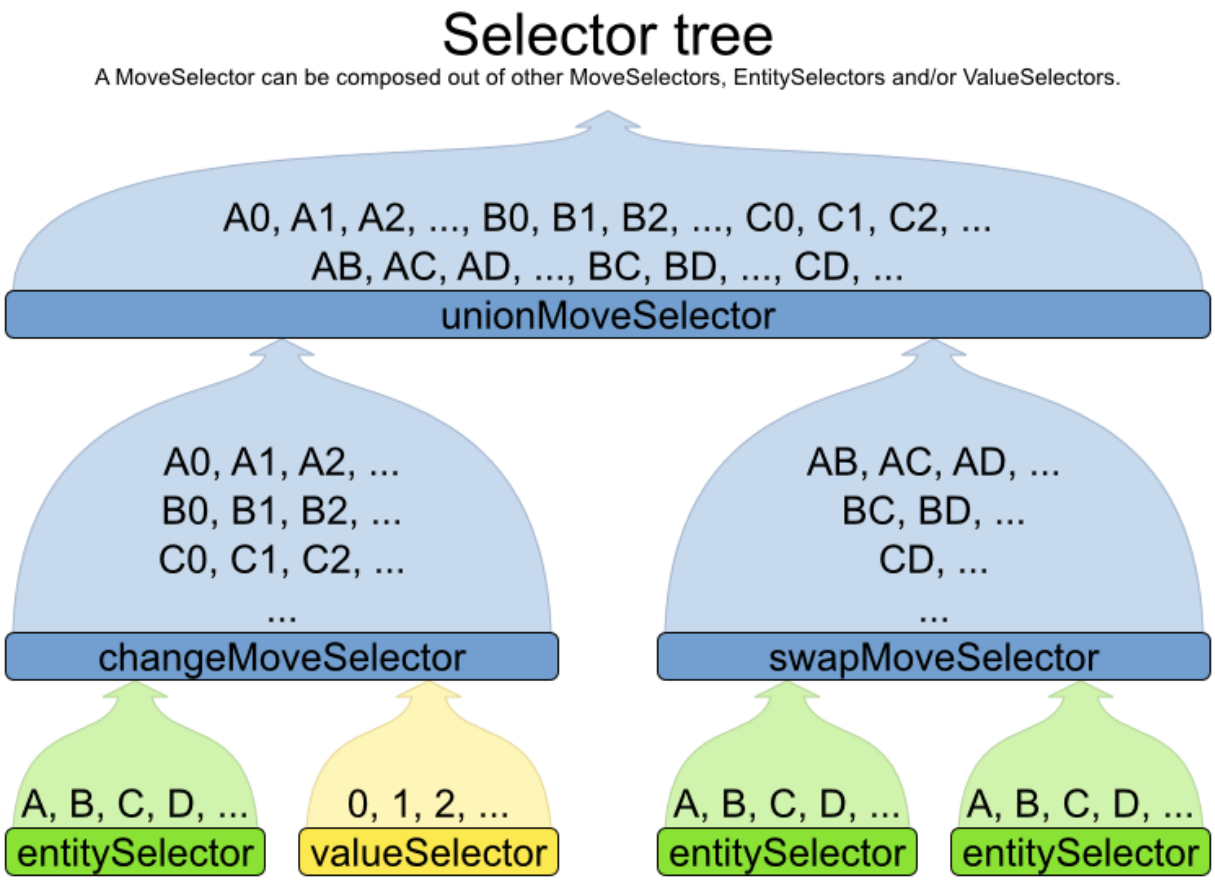
MoveSelector は 、**Move** を作成するために、移動するプランニングエンティティーまたは計画値を 1 つ以上選択する必要があります。**MoveSelectors** と同様、**EntitySelectors** および **ValueSelectors** は、同様の機能セット (スケーラブルな ジャストインタイム選択など) に対応する必要があります。したがって、このすべてに共通する **Selector** インターフェースを実装し、同じように設定します。

MoveSelector は、多くの場合、**EntitySelectors**、**ValueSelectors**、その他の **MoveSelectors** で構成されますが、各セレクターは、必要に応じて個別に設定できます。

```
<unionMoveSelector>
  <changeMoveSelector>
    <entitySelector>
      ...
    </entitySelector>
```

```
      <valueSelector>
        ...
      </valueSelector>
      ...
    </changeMoveSelector>
    <swapMoveSelector>
      ...
    </swapMoveSelector>
  </unionMoveSelector>
```

まとめて、この構造が **Selector** ツリーを形成します。

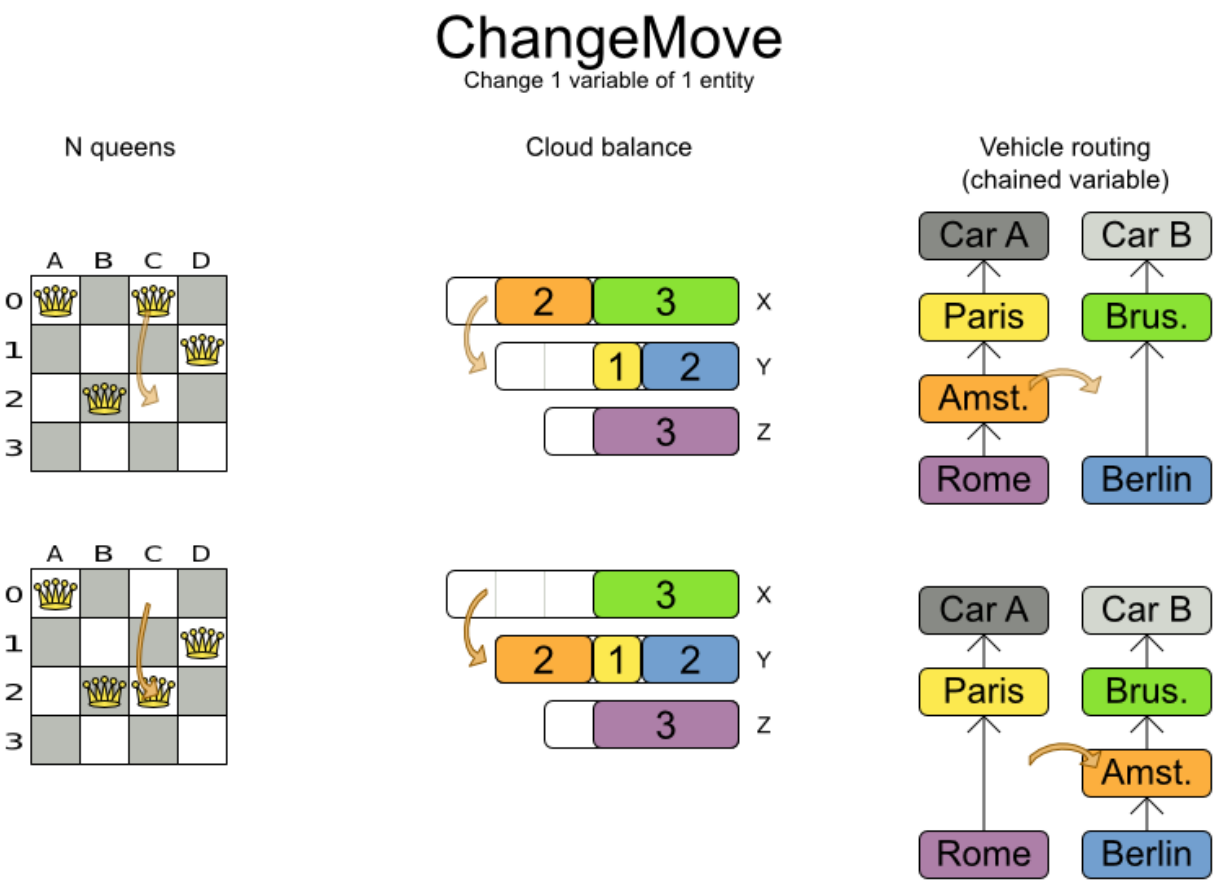


このツリーのルートは **MoveSelector** で、ステップ毎に (部分的に) 反復する最適化アルゴリズムの実装に注入します。

7.2. 一般的な MOVESELECTOR

7.2.1. changeMoveSelector

ChangeMove は、1 つのプランニング変数に対してプランニングエンティティーを 1 つ、計画値を 1 つ選択し、エンティティーの変数をその値に割り当てます。



最も簡単な設定方法:

```
<changeMoveSelector/>
```

1つのエンティティークラスに、複数のエンティティークラスまたは複数のプランニング変数がある場合は、1つの設定が、自動的にすべてのプランニング変数に対する **ChangeMove** の 共用体 に展開されます。

高度な設定方法:

```
<changeMoveSelector>
  ... <!-- Normal selector properties -->
  <entitySelector>
    <entityClass>...Lecture</entityClass>
    ...
  </entitySelector>
  <valueSelector>
    <variableName>room</variableName>
    ...
    <nearbySelection>...</nearbySelection>
  </valueSelector>
</changeMoveSelector>
```

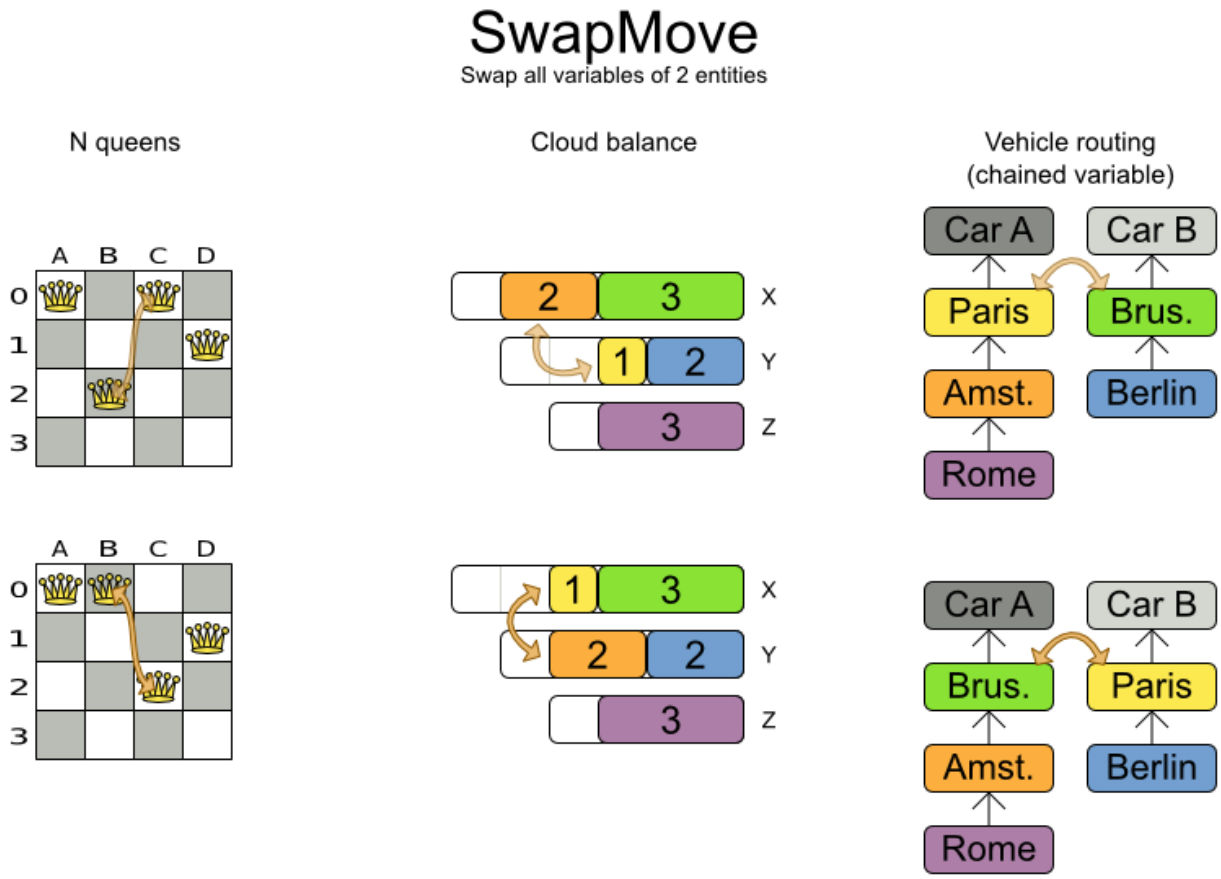
ChangeMove は、粒度が最も細かい Move となります。

重要

メタヒューリスティックアルゴリズムに入るほぼすべての **moveSelector** 設定に、changeMoveSelector またはカスタムの実装が含まれるはずです。これにより、複数の Move を順番に適用して、考えられるすべての 解 (ソリューション) に確実に到達できます (スコアトラップ は考慮しない)。当然ながら、通常ではより粗い他の Move セレクターと結合されます。

7.2.2. swapMoveSelector

SwapMove は、2 つの異なるプランニングエンティティーを選択し、すべてのプランニング変数の計画値を交換します。



1 つの変数における **SwapMove** は、基本的に 2 つの **ChangeMove** になりますが、そのうちの最初の **ChangeMove** が勝利ステップにならないことがしばしばあります。なぜなら、このソリューションでは、ハード制約に違反した状態になるためです。たとえば、両方の授業が同じ部屋で、ハード制約に違反している場合に、2 つの授業の部屋を交換しても、ソリューションは中間状態にはなりません。

最も簡単な設定方法:

```
<swapMoveSelector/>
```

エンティティークラスが複数ある場合は、1 つの設定が、すべてのエンティティークラスに対して、**SwapMove** セレクターの 結合体 に自動的に展開されます。

高度な設定方法:

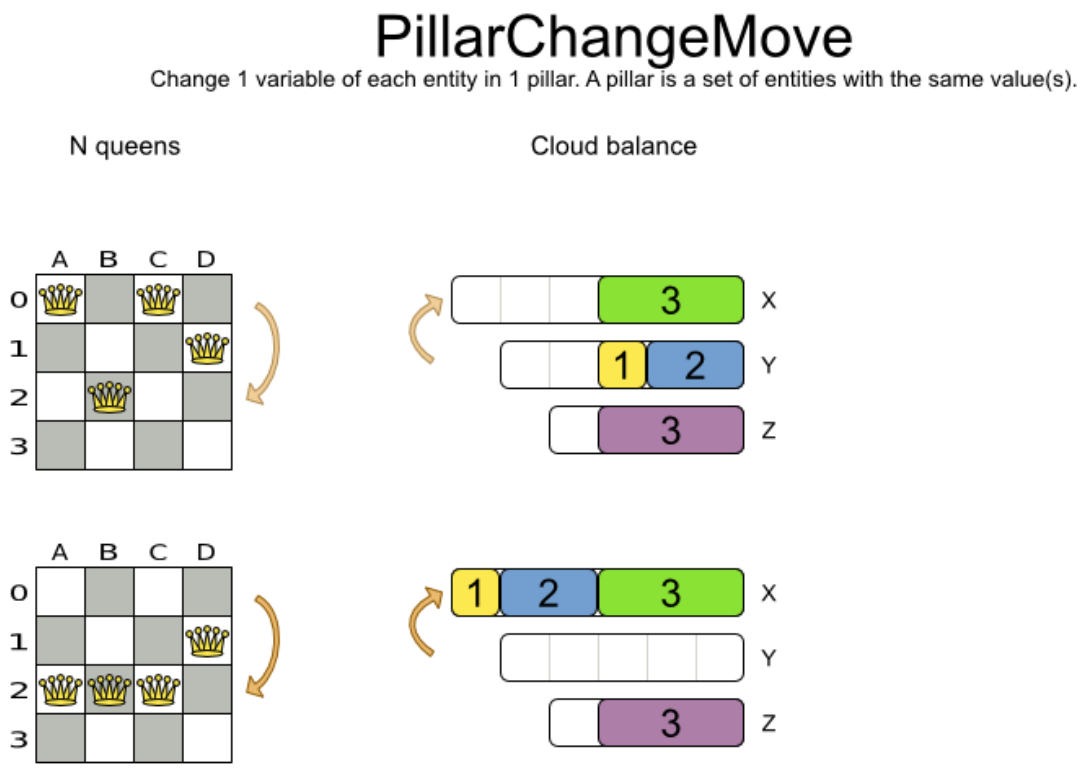
```
<swapMoveSelector>
  ... <!-- Normal selector properties -->
  <entitySelector>
    <entityClass>...Lecture</entityClass>
    ...
  </entitySelector>
  <secondaryEntitySelector>
    <entityClass>...Lecture</entityClass>
    ...
    <nearbySelection>...</nearbySelection>
  </secondaryEntitySelector>
  <variableNameInclude>room</variableNameInclude>
  <variableNameInclude>...</variableNameInclude>
</swapMoveSelector>
```

secondaryEntitySelector が必要になることはほとんどありません。指定しない場合は、同じ **entitySelector** のエンティティーが交換されます。

variableNameInclude プロパティーを 1 つ以上指定すると、すべてのプランニング変数が交換されるわけではなく、指定したものだけが交換されます。たとえば、「コースの時間割」の例では、**variableNameInclude** の部屋だけを指定し、期間ではなく、部屋だけを交換します。

7.2.3. pillarChangeMoveSelector

pillar は、プランニング変数に対して同じ計画値を持つ、プランニングエンティティーのセットです。**PillarChangeMove** は、エンティティーのピラーを 1 つ (もしくはそのサブセット) を選択し、1 つの変数の値 (すべてのエンティティーに同じ) を、別の値に変更します。



上の例では、クイーン A とクイーン C には同じ値 (行 0) があり、行 2 に移動しました。また、黄色と青のプロセスには同じ値 (コンピューター Y) があり、コンピューター X に移動しました。

最も簡単な設定方法:

```
<pillarChangeMoveSelector/>
```

高度な設定方法:

```
<pillarSwapMoveSelector>
  ... <!-- Normal selector properties -->
  <pillarSelector>
    <entitySelector>
      <entityClass>...Lecture</entityClass>
      ...
    </entitySelector>
    <subPillarEnabled>true</subPillarEnabled>
    <minimumSubPillarSize>1</minimumSubPillarSize>
    <maximumSubPillarSize>1000</maximumSubPillarSize>
  </pillarSelector>
  <valueSelector>
    <variableName>room</variableName>
    ...
  </valueSelector>
</pillarSwapMoveSelector>
```

サブピラーは、その変数に対して同じ値を共有するエンティティーのサブセットです。たとえば、クイーン A、クイーン B、クイーン C、クイーン D がすべて行 0 に置かれている場合、そのクイーンはピラーであり、**[A, D]** は多くのサブピラーの 1 つになります。**subPillarEnabled** (デフォルトは **true**) が false の場合、サブピラーは選択されません。サブピラーが有効になると、ピラーそのものも含まれ、**minimumSubPillarSize** プロパティー (デフォルトは **1**) と **maximumSubPillarSize** プロパティー (デフォルトは **infinity**) が、選択した (サブ) ピラーのサイズを制限します。

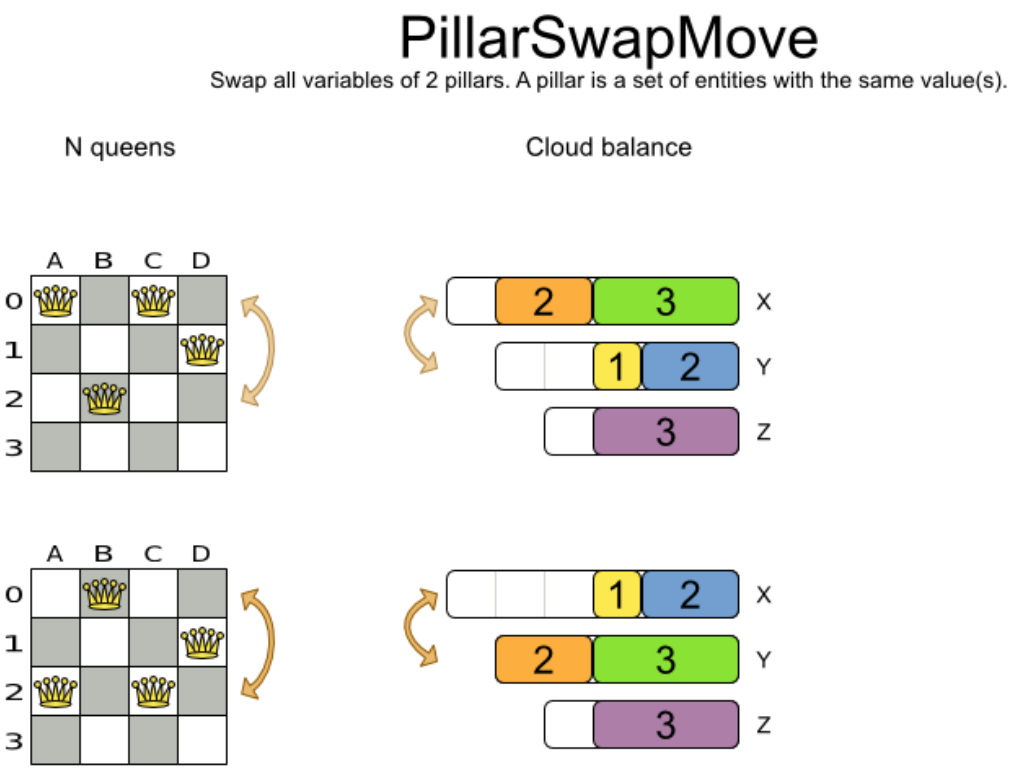
注記

ピラーのサブピラーの数は、ピラーのサイズの指数です。たとえば、ピラーのサイズが 32 の場合のサブピラーは **(2^{32} - 1)** です。したがって、**pillarSelector** は JIT ランダム選択 (デフォルト) だけをサポートします。

その他のプロパティーは、changeMoveSelector で説明します。

7.2.4. pillarSwapMoveSelector

pillar は、プランニング変数に対して同じ計画値を持つ、プランニングエンティティーのセットです。**PillarSwapMove** は、2 つの異なるエンティティーピラーを選択し、そのすべてのエンティティーに対する、すべての変数の値を交換します。



最も簡単な設定方法:

```
<pillarSwapMoveSelector/>
```

高度な設定方法:

```
<pillarSwapMoveSelector>
  ... <!-- Normal selector properties -->
  <pillarSelector>
    <entitySelector>
      <entityClass>...Lecture</entityClass>
      ...
    </entitySelector>
    <subPillarEnabled>true</subPillarEnabled>
    <minimumSubPillarSize>1</minimumSubPillarSize>
    <maximumSubPillarSize>1000</maximumSubPillarSize>
  </pillarSelector>
  <secondaryPillarSelector>
    <entitySelector>
      ...
    </entitySelector>
    ...
  </secondaryPillarSelector>
  <variableNameInclude>room</variableNameInclude>
  <variableNameInclude>...</variableNameInclude>
</pillarSwapMoveSelector>
```

secondaryPillarSelector が必要になることはほとんどありません。指定していない場合は、同じ **pillarSelector** のエンティティーが交換されます。

その他のプロパティーは、swapMoveSelector および pillarChangeMoveSelector で説明しています。

7.2.5. tailChainSwapMoveSelector または 2-opt (連鎖変数のみ)

tailChain は、連鎖のプランニング変数を持つプランニングエンティティーのセットです。この変数は、連鎖の最後の部分を形成します。**tailChainSwapMove** は、テール連鎖を選択し、(同じまたは別のアンカー連鎖にある) 別の計画値のテール連鎖と交換します。対象となる計画値にテール連鎖がない場合は、交換は行われません (その結果、変更のような Move が発生します)。これが、同じアンカー連鎖内で起きた場合は、部分的に連鎖の逆転が発生します。学術論文では、これは、しばしば 2-opt Move と呼ばれます。

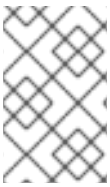
最も簡単な設定方法:

```
<tailChainSwapMoveSelector/>
```

高度な設定方法:

```
<subChainChangeMoveSelector>
  ... <!-- Normal selector properties -->
  <entitySelector>
    <entityClass>...Customer</entityClass>
    ...
  </entitySelector>
  <valueSelector>
    <variableName>previousStandstill</variableName>
    ...
    <nearbySelection>...</nearbySelection>
  </valueSelector>
</subChainChangeMoveSelector>
```

entitySelector は、移動するテール連鎖の開始を選択します。valueSelector は、テール連鎖の移動先を選択します。それ自身にテール連鎖がある場合は、元のテール連鎖の場所に移動します。これは、**secondaryEntitySelector** の代わりに **valueSelector** を使用して、可能な 2-opt Move (たとえばテールの最後に移動) をすべて含み、近傍選択 と適切に連鎖するようにします (不均衡な距離と、交換したエンティティーの距離により、選択の確率は正しくなくなります)。



注記

subChainChangeMoveSelector および **subChainSwapMoveSelector** には、ほぼすべての可能な **tailChainSwapMove** が含まれますが、実験の結果、**tailChainSwapMoves** に重点を置くと効率が上がります。

7.2.6. subChainChangeMoveSelector (連鎖変数のみ)

subChain は、連鎖の一部を形成する連鎖付きプランニング変数がある、一連のプランニングエンティティーになります。**subChainChangeMoveSelector** は subChain を選択し、(同じまたは別のアンカー連鎖にある) 別の場所に動かします。

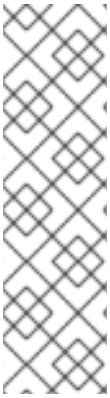
最も簡単な設定方法:

```
<subChainChangeMoveSelector/>
```

高度な設定方法:

```
<subChainChangeMoveSelector>
  ... <!-- Normal selector properties -->
  <entityClass>...Customer</entityClass>
  <subChainSelector>
    <valueSelector>
      <variableName>previousStandstill</variableName>
      ...
    </valueSelector>
    <minimumSubChainSize>2</minimumSubChainSize>
    <maximumSubChainSize>40</maximumSubChainSize>
  </subChainSelector>
  <valueSelector>
    <variableName>previousStandstill</variableName>
    ...
  </valueSelector>
  <selectReversingMoveToo>true</selectReversingMoveToo>
</subChainChangeMoveSelector>
```

subChainSelector が多数のエンティティー (**minimumSubChainSize** (デフォルトは **1**) より多く、**maximumSubChainSize** (デフォルトは **infinity**) より少ない) を選択します。



注記

minimumSubChainSize が **1** (デフォルト) の場合に、(デフォルトでは、各 Move タイプ の選択の可能性は同じで (すべての Move インスタンスではない)、**ChangeMove** インスタンスよりも **SubChainChangeMove** インスタンスの方がはるかに多いため)、このセレクターは選択の可能性ははるかに低いですが、**ChangeMoveSelector** と同じ Move を選択する可能性があります。ただし、実験の結果、**ChangeMoves** にフォーカスすると良いことが分かっているので、**ChangeMoveSelector** だけを削除しないでください。

さらに、**SubChainSwapMoveSelector** に **minimumSubChainSize** を設定すると、サイズが **1** のサブ連鎖を、**2** 以上のサブ連鎖と交換しなくなります。

selectReversingMoveToo プロパティー (デフォルトは true) は、すべてのサブ連鎖の反対を選択することを有効にします。

7.2.7. subChainSwapMoveSelector (連鎖変数のみ)

subChainSwapMoveSelector は 2 つの異なるサブ連鎖を選択し、同一または異なるアンカー連鎖の別の場所に動かします。

最も簡単な設定方法:

```
<subChainSwapMoveSelector/>
```

高度な設定方法:

```
<subChainSwapMoveSelector>
  ... <!-- Normal selector properties -->
  <entityClass>...Customer</entityClass>
  <subChainSelector>
    <valueSelector>
```

```
      <variableName>previousStandstill</variableName>
      ...
    </valueSelector>
    <minimumSubChainSize>2</minimumSubChainSize>
    <maximumSubChainSize>40</maximumSubChainSize>
  </subChainSelector>
  <secondarySubChainSelector>
    <valueSelector>
      <variableName>previousStandstill</variableName>
      ...
    </valueSelector>
    <minimumSubChainSize>2</minimumSubChainSize>
    <maximumSubChainSize>40</maximumSubChainSize>
  </secondarySubChainSelector>
  <selectReversingMoveToo>true</selectReversingMoveToo>
</subChainSwapMoveSelector>
```

secondarySubChainSelector が必要になることはほとんどありません。指定していない場合は、同じ **subChainSelector** のエンティティーを交換します。

その他のプロパティーについては subChainChangeMoveSelector で説明しています。

7.3. 複数の MOVESELECTOR の組み合わせ

7.3.1. unionMoveSelector

unionMoveSelector は、次の **Move** を提供する **MoveSelector** の子を 1 つ選択して **Move** を選択します。

最も簡単な設定方法:

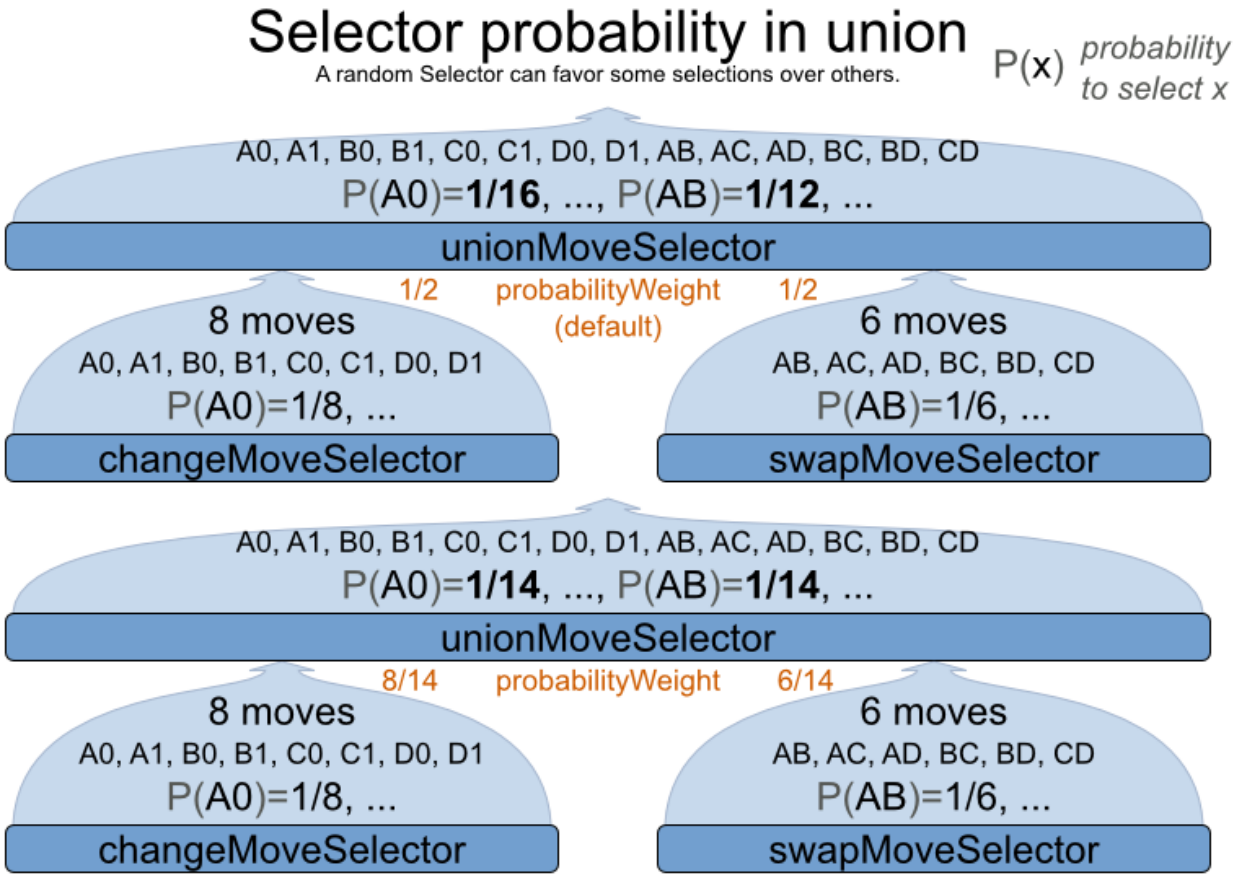
```
<unionMoveSelector>
  <...MoveSelector/>
  <...MoveSelector/>
  <...MoveSelector/>
  ...
</unionMoveSelector>
```

高度な設定方法:

```
<unionMoveSelector>
  ... <!-- Normal selector properties -->

<selectorProbabilityWeightFactoryClass>...ProbabilityWeightFactory</selectorProbabilityWeightFactoryClass>
  <changeMoveSelector>
    <fixedProbabilityWeight>...</fixedProbabilityWeight>
    ...
  </changeMoveSelector>
  <swapMoveSelector>
    <fixedProbabilityWeight>...</fixedProbabilityWeight>
    ...
  </swapMoveSelector>
  <...MoveSelector>
```

```
    <fixedProbabilityWeight>...</fixedProbabilityWeight>
    ...
  </...MoveSelector>
  ...
</unionMoveSelector>
```

selectorProbabilityWeightFactory が、**selectionOrder RANDOM** で、次の Move を提供するために **MoveSelector** の子が選択される頻度を決定します。デフォルトでは、**MoveSelector** の子がそれぞれ選択される可能性は同じになります。



このような子の **fixedProbabilityWeight** を選択する頻度を上げます。たとえば、**unionMoveSelector** が **SwapMove** を返す頻度を **ChangeMove** の 2 倍にします。

```
<unionMoveSelector>
  <changeMoveSelector>
    <fixedProbabilityWeight>1.0</fixedProbabilityWeight>
    ...
  </changeMoveSelector>
  <swapMoveSelector>
    <fixedProbabilityWeight>2.0</fixedProbabilityWeight>
    ...
  </swapMoveSelector>
</unionMoveSelector>
```

可能な **ChangeMove** の数は、可能な **SwapMove** の数とは大きく異なり、さらに、これは問題に依存しています。(各 **MoveSelector** とは異なり) 各 **Move** に同じ選択可能性を付与するには、**FairSelectorProbabilityWeightFactory** を使用します。

```
    <unionMoveSelector>

<selectorProbabilityWeightFactoryClass>org.optaplanner.core.impl.heuristic.selector.common.decorato
r.FairSelectorProbabilityWeightFactory</selectorProbabilityWeightFactoryClass>
      <changeMoveSelector/>
      <swapMoveSelector/>
    </unionMoveSelector>
```

7.3.2. cartesianProductMoveSelector

cartesianProductMoveSelector は新しい **CompositeMove** を選択します。これにより、その **CompositeMove** を構築します。これは、**MoveSelector** の子ごとに **Move** を 1 つ選択して **CompositeMove** に追加します。

最も簡単な設定方法:

```
    <cartesianProductMoveSelector>
      <...MoveSelector/>
      <...MoveSelector/>
      <...MoveSelector/>
      ...
    </cartesianProductMoveSelector>
```

高度な設定方法:

```
    <cartesianProductMoveSelector>
      ... <!-- Normal selector properties -->
      <ignoreEmptyChildIterators>true</ignoreEmptyChildIterators>
      <changeMoveSelector>
        ...
      </changeMoveSelector>
      <swapMoveSelector>
        ...
      </swapMoveSelector>
      <...MoveSelector>
        ...
      </...MoveSelector>
      ...
    </cartesianProductMoveSelector>
```

ignoreEmptyChildIterators プロパティー (デフォルトでは true) は、何らかの move が返されるように、空の **childMoveSelector** をすべて無視します。たとえば、**changeMoveSelector** の A および B のデカルト積で、(すべてのエンティティーが動かせないため) B が空であれば、**ignoreEmptyChildIterators** が **false** の場合は Move を返さず、**ignoreEmptyChildIterators** が **true** の場合は A の Move を返します。

その 2 つの子セレクターが、同じエンティティーまたは値を効果的に使用するには、 Move フィルタリングではなく、模倣選択 を使用します。

7.4. ENTITYSELECTOR

最も簡単な設定方法:

```
    <entitySelector/>
```

高度な設定方法:

```
<entitySelector>
  ... <!-- Normal selector properties -->
  <entityClass>org.optaplanner.examples.curriculumcourse.domain.Lecture</entityClass>
</entitySelector>
```

entityClass プロパティーには、エンティティークラスが複数あるため、自動的に推測できない場合にのみ必要になります。

7.5. VALUESELECTOR

最も簡単な設定方法:

```
<valueSelector/>
```

高度な設定方法:

```
<valueSelector>
  ... <!-- Normal selector properties -->
  <variableName>room</variableName>
</valueSelector>
```

variableName プロパティーは、(関連するエンティティークラスに対して) 変数が複数あるため、自動的に推測できない場合に限り必要になります。

新型の構築ヒューリスティック設定において、**EntitySelector** の **entityClass** のダウンキャストが必要になる場合があります。これは **downcastEntityClass** プロパティーから行えます。

```
<valueSelector>
  <downcastEntityClass>...LeadingExam</downcastEntityClass>
  <variableName>period</variableName>
</valueSelector>
```

選択したエンティティーをダウンキャストできない場合は、そのエンティティーに対して **ValueSelector** が空になります。

7.6. SELECTOR の一般的な機能

7.6.1. CacheType: Create Moves Ahead of Time または Just In Time

セレクター の **cacheType** は、選択 (**Move**、エンティティー、値など) がいつ作成され、どのぐらい存続するかを決定します。

ほぼすべての セレクター で **cacheType** の設定に対応しています。

```
<changeMoveSelector>
  <cacheType>PHASE</cacheType>
  ...
</changeMoveSelector>
```

次の **cacheTypes** はサポートされます。

- **JUST_IN_TIME** (デフォルト): キャッシュは作成されません。各選択 (**Move** など) を使用する直前に構築します。これにより、メモリーフットプリントが十分な大きさになります。
- **STEP**: キャッシュされます。ステップの初めに各選択 (**Move** など) を作成し、残りのステップに使用するために、その選択をリストにキャッシュします。これにより、メモリーフットプリントが著しく大きくなります。
- **PHASE**: キャッシュされます。Solver フェーズの初めに各選択 (**Move** など) を作成し、残りのステップに使用するために、その選択をリストにキャッシュします。一部の選択では、各ステップでリストが変更になるため、フェーズをキャッシュすることはできません。これにより、メモリーフットプリントが著しく大きくなりますが、パフォーマンスがわずかに向上します。
- **SOLVER**: キャッシュされます。**Solver** の初めに各選択(**Move** など) を作成し、残りの **Solver** 用に、リストにキャッシュします。各ステップでリストが変更になるため、一部の選択では Solver のキャッシュを取得することができません。これにより、メモリーフットプリントが著しく大きくなりますが、パフォーマンスがわずかに向上します。

また、複合セレクターに **cacheType** を設定できます。

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <changeMoveSelector/>
  <swapMoveSelector/>
  ...
</unionMoveSelector>
```

cacheType を高くした場合を除いて、キャッシュ済セレクターにネストされたセレクターをキャッシュするように設定することができません。たとえば、ステップ をキャッシュする **unionMoveSelector** は、フェーズ をキャッシュする **changeMoveSelector** を保持しますが、ステップをキャッシュする **changeMoveSelector** は保持しません。

7.6.2. SelectionOrder: Original、Sorted、Random、Shuffled、または Probabilistic

セレクター の **selectionOrder** は、選択 (**Moves**、エンティティー、値など) が反復する順番を決定します。最適化アルゴリズムは、通常、はじめから、**MoveSelector** の選択のサブセットを通してのみ反復しますが、**selectionOrder** が、実際に評価する **Move** を決定することが重要になります。

ほぼすべての セレクター は、**selectionOrder** の設定をサポートします。

```
<changeMoveSelector>
  ...
  <selectionOrder>RANDOM</selectionOrder>
  ...
</changeMoveSelector>
```

以下の **selectionOrders** がサポートされます。

- **ORIGINAL**: デフォルトの順に、選択項目 (**Moves**、エンティティー、値など) が選択されます。各選択項目が選択されるのは、一度だけです。
  - 例: A0、A1、A2、A3...、B0、B1、B2、B3...、C0、C1、C2、C3... です。
- **SORTED**: ソートした順番に、選択項目 (**Moves**、エンティティー、値など) が選択されます。

各選択項目が選択されるのは、一度だけです。**cacheType ≥ STEP** は必須で、多くの場合、構築ヒューリスティックの **entitySelector** または **valueSelector** で使用されます。ソートされた選択 を参照してください。

  - 例: A0、B0、C0...、A2、B2、C2...、A1、B1、C1...

- **RANDOM** (デフォルト): シャッフルしていないランダムの順番で、選択項目 (**Moves**、エンティティー、値など) が選択されます。各選択項目は複数回選択できます。キャッシュを必要としないため、パフォーマンスが十分に上がります。

  - 例: C2、A3、B1、C2、A0、C0...

- **SHUFFLED**: シャッフルしたランダム順で、選択項目 (**Moves**、エンティティー、値など) が選択されます。各選択項目が選択されるのは一度だけです。**cacheType ≥ STEP** が必要です。これにより、パフォーマンスが著しく上がりますが、キャッシュが必須であるだけでなく、選択されてなくても、各要素に対して乱数が生成されているためです (これは、拡大するときの主要な大多数になります)。

  - 例: C2、A3、B1、A0、C0...

- **PROBABILISTIC**: 各要素の選択可能性に基づいて、ランダム順で選択項目 (**Moves**、エンティティー、値など) が選択されます。可能性が高い選択項目は、可能性が低い要素よりも選択される可能性が高くなります。選択項目は、複数回選択される可能性があります。**cacheType ≥ STEP** が必要です。ほとんどの場合、**entitySelector** または **valueSelector** で使用されます。確立的選択 を参照してください。

  - 例: B1、B1、A1、B2、B1、C2、B1、B1...

selectionOrder は、複合セレクターにも設定できます。

注記

セレクター がキャッシュされると、そのネストされたすべての セレクター は、必然的に **selectionOrder ORIGINAL** にデフォルト設定されます。そのネストされた セレクター の **selectionOrder** を上書きしないようにします。

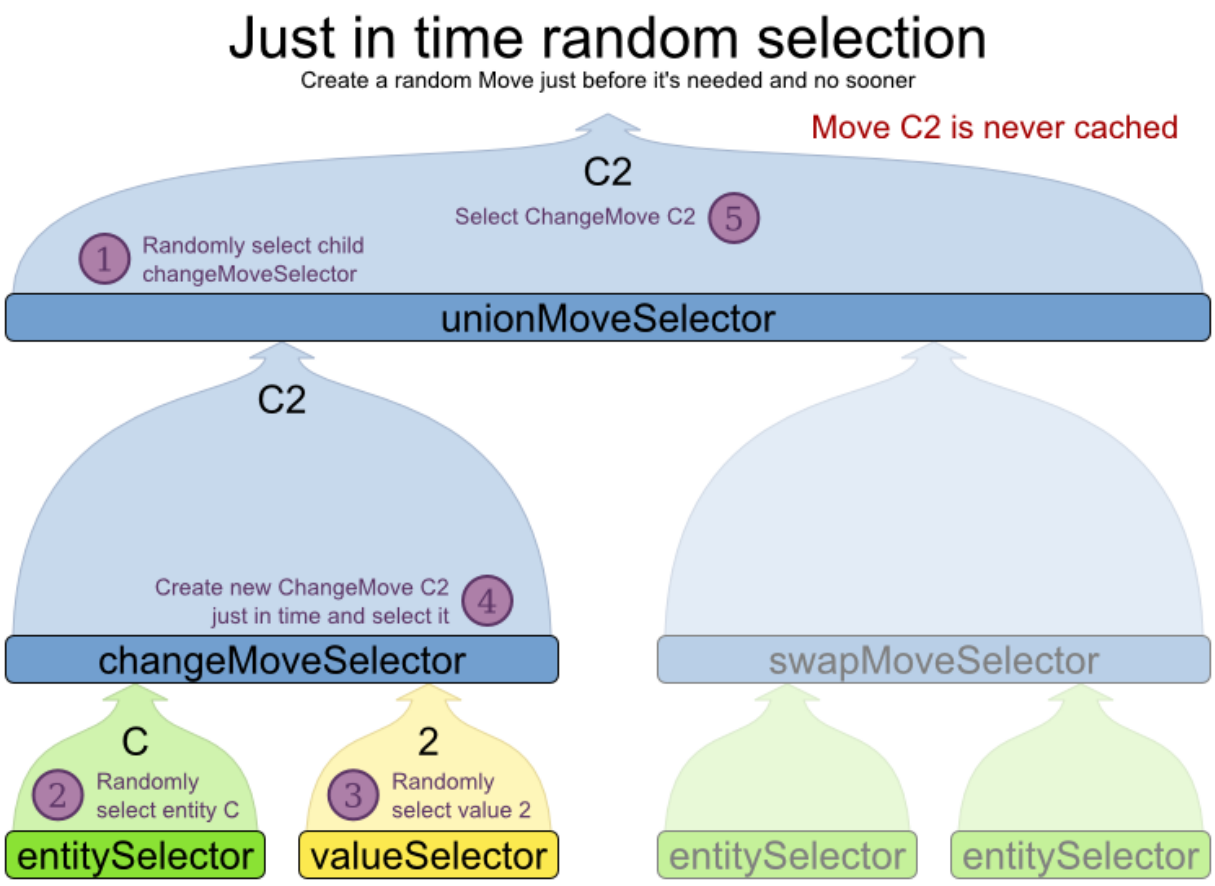
7.6.3. CacheType と SelectionOrder で推奨される組み合わせ

7.6.3.1. Just in Time ランダム選択 (デフォルト)

この組み合わせは、メモリーフットプリントおよびパフォーマンスが十分に増えるため、ユースケースが大きい (10 000 以上のエンティティー) 場合に重要になります。サイズがこのぐらいになると、他の組み合わせでは実行できない場合もしばしばあります。また、サイズがこれよりも小さいユースケースでも有効なので、試しに使用するのに適しています。これはデフォルトであるため、実際には、**cacheType** および **selectionOrder** を明示的に設定することはなくなりました。

```
<unionMoveSelector>
  <cacheType>JUST_IN_TIME</cacheType>
  <selectionOrder>RANDOM</selectionOrder>

  <changeMoveSelector/>
  <swapMoveSelector/>
</unionMoveSelector>
```

この動作は次のようになります。**Iterator<Move>.next()** が呼び出されると、子の **MoveSelector** がランダムに選択され (1)、それが **Move** (2、3、4) をランダムに作成し、返されます (5)。



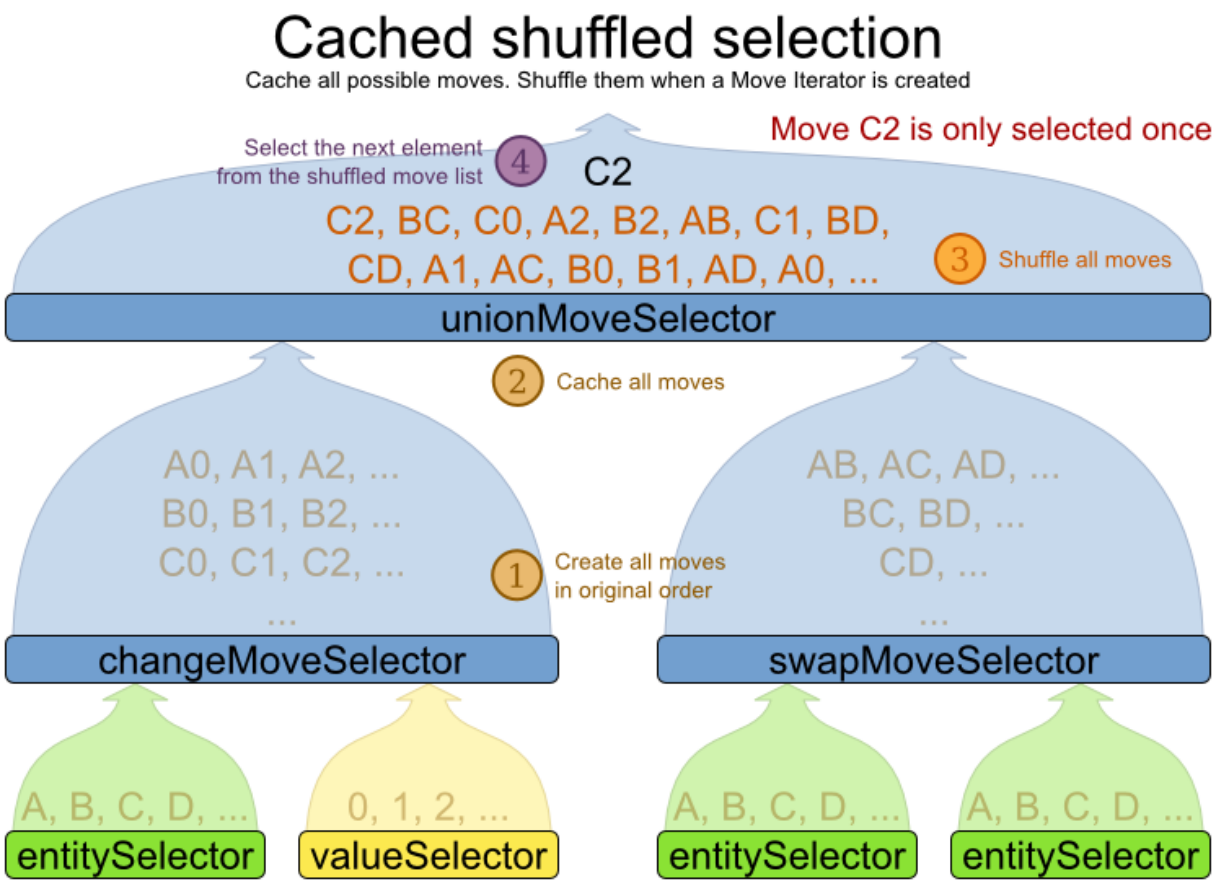
Move のリストは作成されず、実際に選択された **Move** に対する乱数を生成します。

7.6.3.2. キャッシュしたシャッフル選択

小規模や中規模のユースケース (5000 エンティティー以下) で、この組み合わせがしばしば勝利します。サイズがこれよりも大きくなると、メモリーフットプリントとパフォーマンスが著しく大きくなります。

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SHUFFLED</selectionOrder>

  <changeMoveSelector/>
  <swapMoveSelector/>
</unionMoveSelector>
```

この動作は次のようにします。フェーズ (または **cacheType** で決まるステップ) の開始時にすべての Move が作成され (1)、キャッシュされます (2)。**MoveSelector.iterator()** が呼び出されると、Move はシャッフルされます (3)。**Iterator<Move>.next()** が呼び出されると、シャッフルされたリストで次の要素が返されます (4)。



各 **Move** は、ランダム順ではありますが、一度だけ選択されます。

(おそらくネストされている) セレクター で ステップ が必要ない場合は、**cacheType** の **PHASE** を使用します。それ以外の場合は、以下のようになります。

```
<unionMoveSelector>
  <cacheType>STEP</cacheType>
  <selectionOrder>SHUFFLED</selectionOrder>

  <changeMoveSelector>
    <cacheType>PHASE</cacheType>
  </changeMoveSelector>
  <swapMoveSelector/>
    <cacheType>PHASE</cacheType>
  </swapMoveSelector>
  <pillarSwapMoveSelector/><!-- Does not support cacheType PHASE -->
</unionMoveSelector>
```

7.6.3.3. キャッシュしたランダム選択

この組み合わせは、中規模のユースケース (特に、焼きなまし法など、ステップが速い最適化アルゴリズム) でしばしば有効です。キャッシュされるシャッフル選択とは異なり、各ステップの初めに Move リストをシャッフルし、時間を無駄にしません。

```
<unionMoveSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>RANDOM</selectionOrder>
```

```
    <changeMoveSelector/>
    <swapMoveSelector/>
  </unionMoveSelector>
```

7.6.4. フィルターをかけた選択

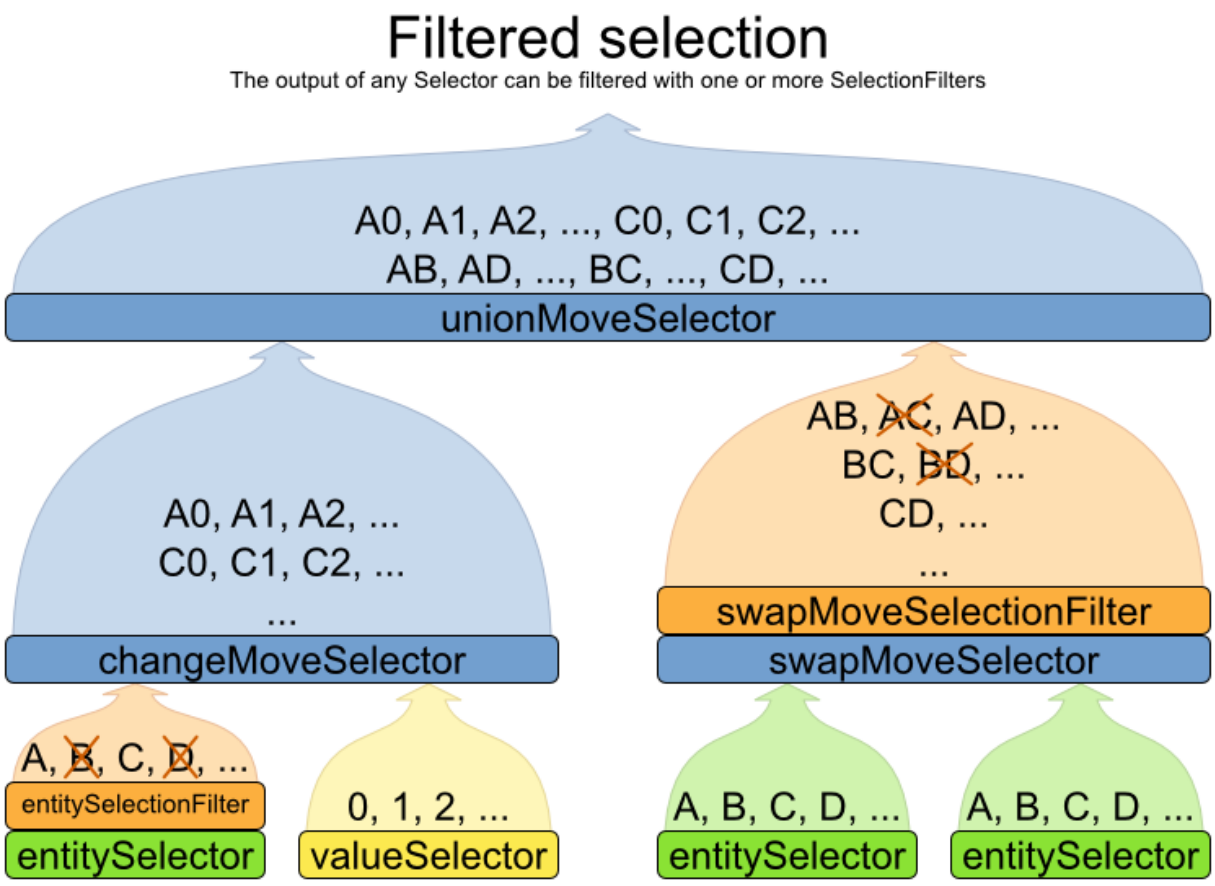
以下の理由により、特定の Move を選択しない場合があります。

- Move が無意味で、CPU 時間を無駄にしているだけの場合。たとえば、同じコースの 2 つの授業を交換すると、1 つのコースの授業はすべて交換できるため、同じスコア、同じスケジュールになります (同じ教師、同じ生徒、同じテーマ)。
- Move を行うと 、組み込みハード制約 が壊れ、ソリューションは実行できなくなりますが、(パフォーマンスを上げるため) スコア関数による組み込みのハード制約の確認はありません。たとえば、体育の授業が体育室以外の部屋には変更されないなどです。

注記

組み込みハード制約は、すべての Solver フェーズのすべての Move タイプに、フィルターを適切に設定する必要があります。たとえば、局所探索法の変更 Move にフィルターを設定する場合は、交換する Move にフィルターを設定する必要があります。この Move では、体育の授業の部屋を、元の部屋が体育室ではない別の授業と交換します。さらに、構築ヒューリスティックの変更 Move にもフィルターを設定する必要があります (これには、高度な設定が必要です)。

セレクターツリーのすべてのセレクター (**MoveSelector**、**EntitySelector**、**ValueSelector** を含む) で、フィルター選択が発生する可能性があります。これは、**cacheType** および **selectionOrder** と連携します。



フィルタリングは、**SelectionFilter** インターフェースを使用します。

```
public interface SelectionFilter<T> {

    boolean accept(ScoreDirector scoreDirector, T selection);

}
```

不要な 選択 に対して **false** を返す **accept** メソッドを実装します。承認されない Move は選択されず、**doMove** メソッドは呼び出されません。

```
public class DifferentCourseSwapMoveFilter implements SelectionFilter<SwapMove> {

    public boolean accept(ScoreDirector scoreDirector, SwapMove move) {
        Lecture leftLecture = (Lecture) move.getLeftEntity();
        Lecture rightLecture = (Lecture) move.getRightEntity();
        return !leftLecture.getCourse().equals(rightLecture.getCourse());
    }

}
```

できる限り低いレベルで、このフィルターを適用します。多くの場合は、関連のエンティティーと値の両方を知る必要があり、**moveSelector** に **filterClass** を適用する必要があります。

```
    <swapMoveSelector>

<filterClass>org.optaplanner.examples.curriculumcourse.solver.move.DifferentCourseSwapMoveFilter
```

```
</filterClass>
  </swapMoveSelector>
```

ただし、可能な場合は、さらに低いレベル (たとえば、**entitySelector** や **valueSelector** の **filterClass**) で適用します。

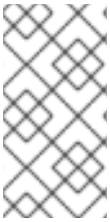
```
<changeMoveSelector>
  <entitySelector>
    <filterClass>...EntityFilter</filterClass>
  </entitySelector>
</changeMoveSelector>
```

1 つのセレクターに、複数の **filterClass** 要素を設定できます。

7.6.5. ソートした選択

ソートした選択は、セレクターツリーにあるすべての セレクター で発生する可能性があります (**MoveSelector**、**EntitySelector**、**ValueSelector** など)。これは、**cacheType JUST_IN_TIME** では使用できませんが、**selectionOrder SORTED** で使用できます。

これは、主に構築ヒューリスティックで使用されています。



注記

選択した構造ヒューリスティックにソートが含まれる場合 (たとえば **FIRST_FIT_DECREASING** では **EntitySelector** がソートされることを示しています) は、ソートを使用して **Selector** を明示的に設定する必要はありません。セレクター を明示的に設定する場合は、構築ヒューリスティックのデフォルト設定が上書きされます。

7.6.5.1. SorterManner でソートした選択

一部の セレクター では、追加設定しなくても **SorterManner** が実装されています。

- **EntitySelector** は以下に対応します。
  - **DECREASING_DIFFICULTY**: プランニングエンティティーの難易度 に従って、プランニングエンティティーを降順でソートします。ドメインモデルに、プランニングエンティティーの難易度をアノテートする必要があります。

```
<entitySelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
  <sorterManner>DECREASING_DIFFICULTY</sorterManner>
</entitySelector>
```

- **ValueSelector** は以下に対応します。
  - **INCREASING_STRENGTH**: プランニングエンティティー値の強度 に従って、計画値を昇順でソートします。ドメインモデルに、計画値の強度をアノテートする必要があります。

```
<valueSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
  <sorterManner>INCREASING_STRENGTH</sorterManner>
</valueSelector>
```

- **DECREASING_STRENGTH**: 計画値の強度に従って計画値を降順でソートします。ドメインモデルに、計画値の強度をアノテートする必要があります。

```
<valueSelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
  <sorterManner>DECREASING_STRENGTH</sorterManner>
</valueSelector>
```

7.6.5.2. Comparator でソートした選択

セレクター は、Plain Old **Comparator** を使用すると、簡単にソートできます。

```
public class CloudProcessDifficultyComparator implements Comparator<CloudProcess> {

    public int compare(CloudProcess a, CloudProcess b) {
        return new CompareToBuilder()
                .append(a.getRequiredMultiplicand(), b.getRequiredMultiplicand())
                .append(a.getId(), b.getId())
                .toComparison();
    }

}
```

また、Comparator を設定する必要があります (ドメインモデルにアノテートされており、最適化アルゴリズムによって自動的に適用される場合は除きます)。

```
<entitySelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
  <sorterComparatorClass>...CloudProcessDifficultyComparator</sorterComparatorClass>
  <sorterOrder>DESCENDING</sorterOrder>
</entitySelector>
```

sorterOrder 要素は任意です。デフォルト値は **ASCENDING** になります。

7.6.5.3. SelectionSorterWeightFactory でソートした選択

ソリューション全体で セレクター をソートする必要がある場合は、代わりに **SelectionSorterWeightFactory** を使用します。

```
public interface SelectionSorterWeightFactory<Sol extends Solution, T> {

    Comparable createSorterWeight(Sol solution, T selection);

}
```

```
public class QueenDifficultyWeightFactory implements SelectionSorterWeightFactory<NQueens,
Queen> {

    public Comparable createSorterWeight(NQueens nQueens, Queen queen) {
        int distanceFromMiddle = calculateDistanceFromMiddle(nQueens.getN(),
```

```
queen.getColumnIndex());
        return new QueenDifficultyWeight(queen, distanceFromMiddle);
    }

    // ...

    public static class QueenDifficultyWeight implements Comparable<QueenDifficultyWeight> {

        private final Queen queen;
        private final int distanceFromMiddle;

        public QueenDifficultyWeight(Queen queen, int distanceFromMiddle) {
            this.queen = queen;
            this.distanceFromMiddle = distanceFromMiddle;
        }

        public int compareTo(QueenDifficultyWeight other) {
            return new CompareToBuilder()
                    // The more difficult queens have a lower distance to the middle
                    .append(other.distanceFromMiddle, distanceFromMiddle) // Decreasing
                    // Tie breaker
                    .append(queen.getColumnIndex(), other.queen.getColumnIndex())
                    .toComparison();
        }

    }

}
```

また、Comparator を設定する必要があります (ドメインモデルにアノテートされており、最適化アルゴリズムによって自動的に適用される場合は除きます)。

```
<entitySelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
  <sorterWeightFactoryClass>...QueenDifficultyWeightFactory</sorterWeightFactoryClass>
  <sorterOrder>DESCENDING</sorterOrder>
</entitySelector>
```

sorterOrder 要素は任意です。デフォルト値は **ASCENDING** になります。

7.6.5.4. SelectionSorter でソートした選択

または、**SelectionSorter** インターフェースを直接使用することもできます。

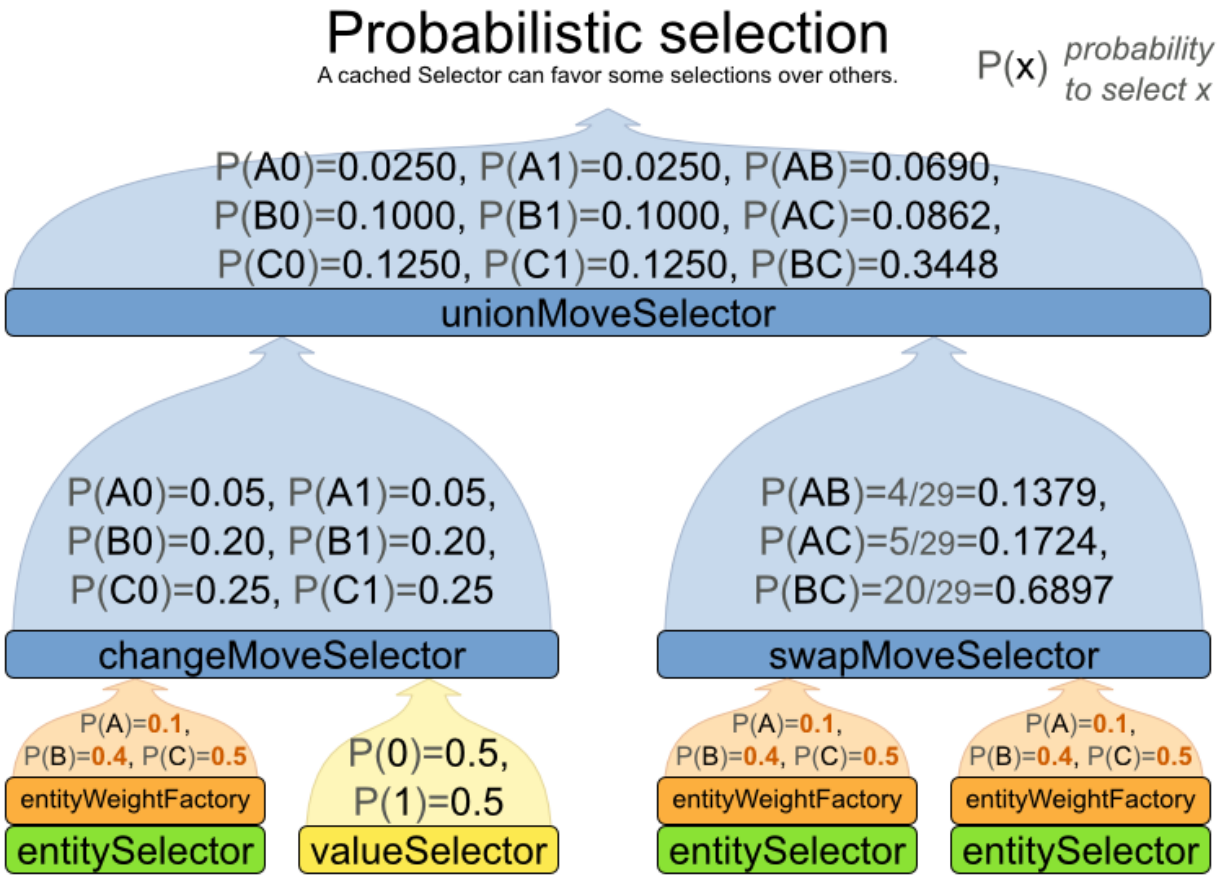
```
public interface SelectionSorter<T> {

    void sort(ScoreDirector scoreDirector, List<T> selectionList);

}
```

```
<entitySelector>
  <cacheType>PHASE</cacheType>
  <selectionOrder>SORTED</selectionOrder>
```

```
    <sorterClass>...MyEntitySorter</sorterClass>
  </entitySelector>
```

7.6.6. 確率的な選択

確率的な選択 は、セレクターツリーのどの セレクター でも発生します。これには、**MoveSelector**、**EntitySelector**、または **ValueSelector** が含まれます。これは、**cacheType JUST_IN_TIME** では使用できず、**selectionOrder PROBABILISTIC** で使用できます。



各選択には **probabilityWeight** があります。これは、選択が行われる可能性を決定します。

```
public interface SelectionProbabilityWeightFactory<T> {

    double createProbabilityWeight(ScoreDirector scoreDirector, T selection);

}
```

```
    <entitySelector>
      <cacheType>PHASE</cacheType>
      <selectionOrder>PROBABILISTIC</selectionOrder>

<probabilityWeightFactoryClass>...MyEntityProbabilityWeightFactoryClass</probabilityWeightFactory
Class>
    </entitySelector>
```

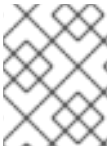
たとえば、プロセス A (**probabilityWeight 2.0**)、プロセス B (**probabilityWeight 0.5**)、プロセス C (**probabilityWeight 0.5**) の 3 つのエンティティーがある場合、プロセス A が選択される回数は、B および C の 4 倍となります。

7.6.7. 制限された選択

特に (acceptedCountLimit に対応しない) 構築ヒューリスティックの場合は、可能な Move をすべて選択しても十分にスケーリングするわけではありません。

各ステップで行う選択の数を制限するには、セレクターに **selectedCountLimit** を適用します。

```
<changeMoveSelector>
  <selectedCountLimit>100</selectedCountLimit>
</changeMoveSelector>
```



注記

局所探索法を増減するには、通常は、**selectedCountLimit** よりも acceptedCountLimit を設定することが推奨されます。

7.6.8. 模倣選択 (記録/再現)

模倣選択時に、1 つの通常のセレクターは、その選択を記録し、他の特別セレクター 1 つまたは複数が、その選択を再現します。記録するセレクターは通常のセレクターとして動作し、その他の設定プロパティーすべてに対応します。再現するセレクターは、記録する選択を模倣し、その他の設定プロパティーはサポートしません。

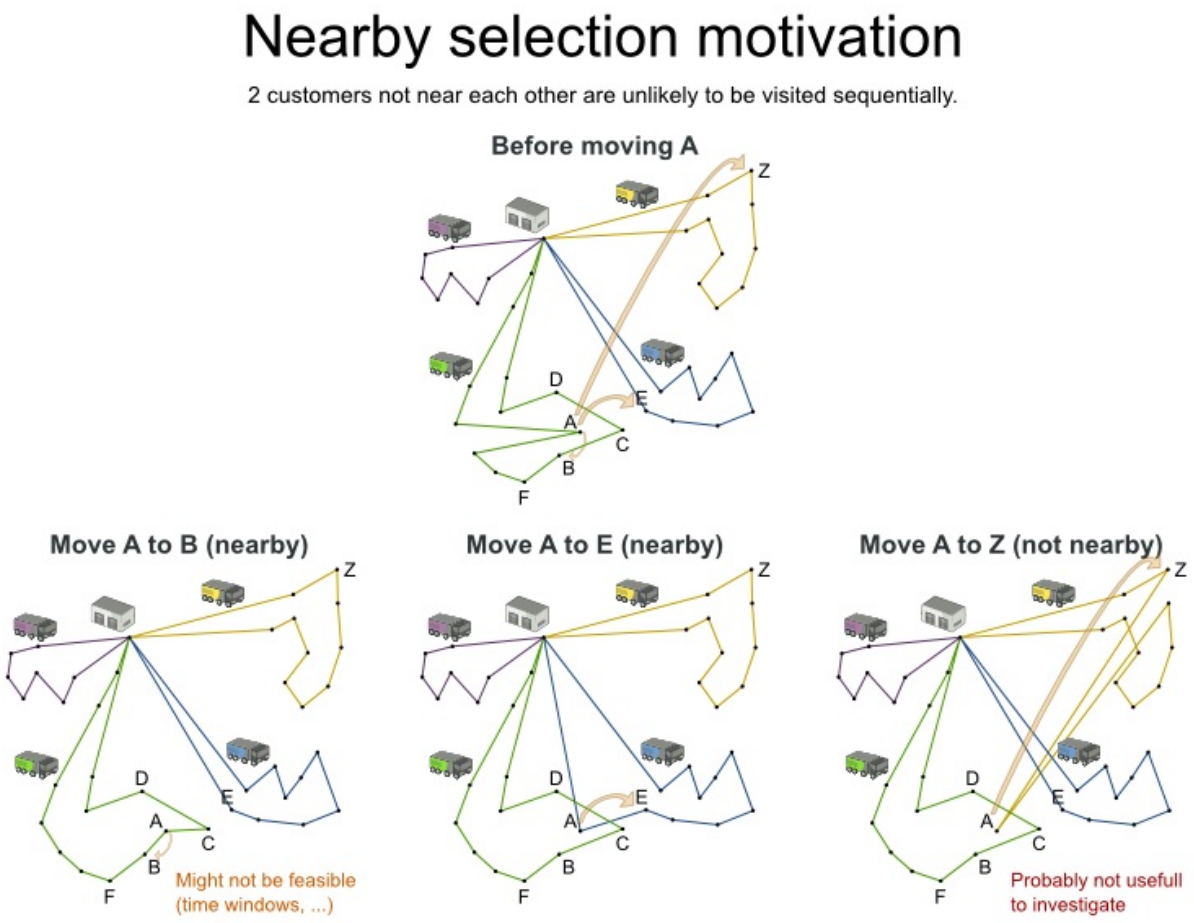
記録するセレクターには ID が必要です。再現するセレクターは、**mimicSelectorRef** で、記録したセレクターの ID を参照する必要があります。

```
<cartesianProductMoveSelector>
  <changeMoveSelector>
    <entitySelector id="entitySelector"/>
    <valueSelector>
      <variableName>period</variableName>
    </valueSelector>
  </changeMoveSelector>
  <changeMoveSelector>
    <entitySelector mimicSelectorRef="entitySelector"/>
    <valueSelector>
      <variableName>room</variableName>
    </valueSelector>
  </changeMoveSelector>
</cartesianProductMoveSelector>
```

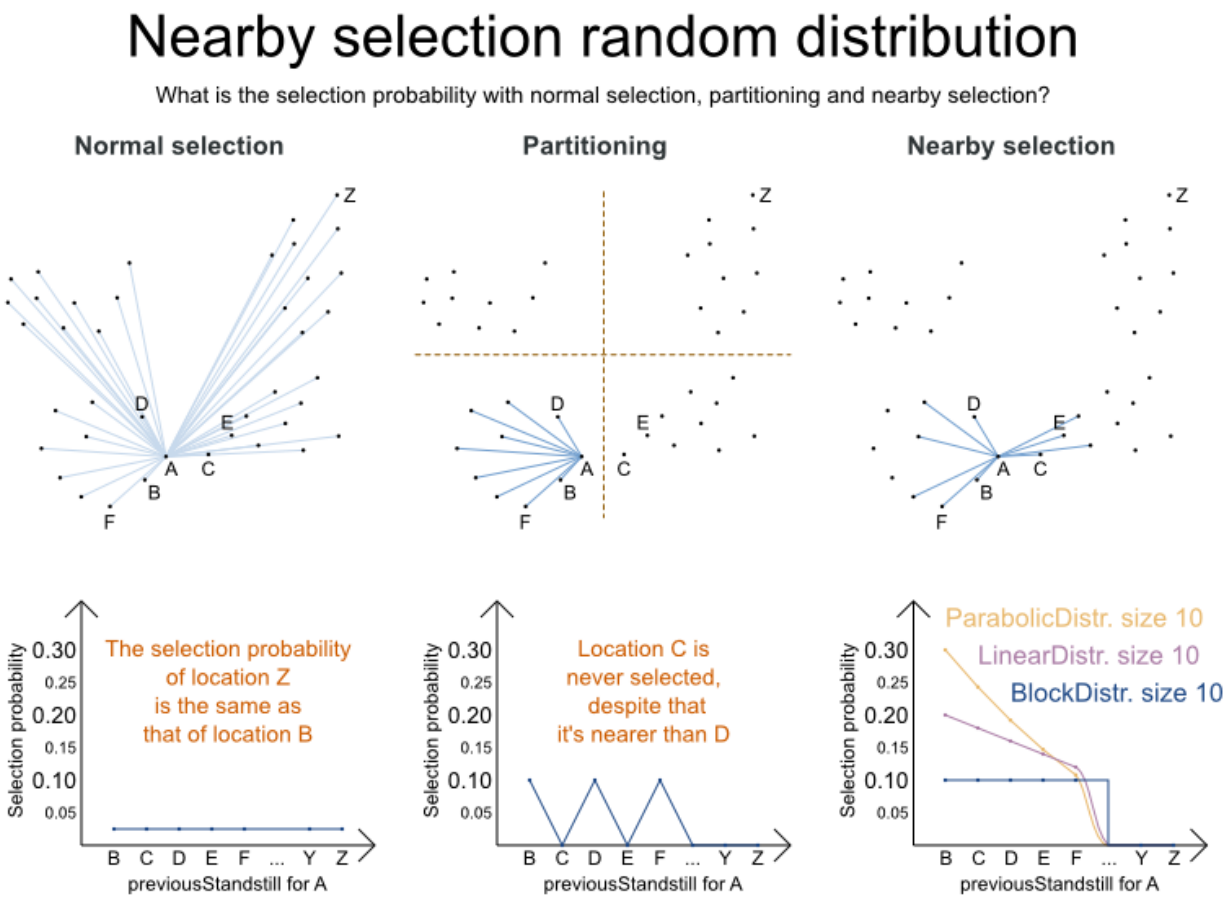
模倣選択は、同じエンティティーに影響する 2 つの Move から 複合 Move を作成するのに便利です。

7.6.9. 近傍選択

ユースケースの中には (TSP や VRP など、そして連鎖がない変数を使用している場合)、エンティティーを近傍値に変更するか、近傍エンティティーを交換すると、スケーラビリティーと、ソリューションの品質が大幅に向上します。



近傍選択は、最初に移動したエンティティーに近いエンティティーまたは値を選択する可能性が上がります。



2 つのエンティティーまたは値の間の距離はドメインによって異なります。したがって、**NearbyDistanceMeter** インターフェースを実装します。

```
public interface NearbyDistanceMeter<O, D> {

    double getNearbyDistance(O origin, D destination);

}
```

これは、距離を示す **double** を返します。

```
public class CustomerNearbyDistanceMeter implements NearbyDistanceMeter<Customer, Standstill>
{

    public double getNearbyDistance(Customer origin, Standstill destination) {
        return origin.getDistanceTo(destination);
    }

}
```

近傍選択を設定するには、**entitySelector** または **valueSelector** に **nearbySelection** 要素を追加し、模倣選択 を使用して、選択から、どのエンティティーが近いかを指定します。

```
    <unionMoveSelector>
      <changeMoveSelector>
        <entitySelector id="entitySelector1"/>
        <valueSelector>
```

```
      <nearbySelection>
        <originEntitySelector mimicSelectorRef="entitySelector1"/>
        <nearbyDistanceMeterClass>...CustomerNearbyDistanceMeter</nearbyDistanceMeterClass>
        <parabolicDistributionSizeMaximum>40</parabolicDistributionSizeMaximum>
      </nearbySelection>
    </valueSelector>
  </changeMoveSelector>
  <swapMoveSelector>
    <entitySelector id="entitySelector2"/>
    <secondaryEntitySelector>
      <nearbySelection>
        <originEntitySelector mimicSelectorRef="entitySelector2"/>
        <nearbyDistanceMeterClass>...CustomerNearbyDistanceMeter</nearbyDistanceMeterClass>
        <parabolicDistributionSizeMaximum>40</parabolicDistributionSizeMaximum>
      </nearbySelection>
    </secondaryEntitySelector>
  </swapMoveSelector>
  <tailChainSwapMoveSelector>
    <entitySelector id="entitySelector3"/>
    <valueSelector>
      <nearbySelection>
        <originEntitySelector mimicSelectorRef="entitySelector3"/>
        <nearbyDistanceMeterClass>...CustomerNearbyDistanceMeter</nearbyDistanceMeterClass>
        <parabolicDistributionSizeMaximum>40</parabolicDistributionSizeMaximum>
      </nearbySelection>
    </valueSelector>
  </tailChainSwapMoveSelector>
</unionMoveSelector>
```

distributionSizeMaximum パラメーターは 1 にすべきではありません。理由は、最も近い値がすでに、現在のエンティティーの計画値である場合に、唯一選択できる Move は実行できないからです。

すべての要素を選択するには、エンティティーの数にかかわらず、ディストリビューションの種類だけを選択します (したがって **distributionSizeMaximum** パラメーターはありません)。

```
  <nearbySelection>

<nearbySelectionDistributionType>PARABOLIC_DISTRIBUTION</nearbySelectionDistributionType>
  </nearbySelection>
```

次の **NearbySelectionDistributionTypes** に対応します。

- **BLOCK_DISTRIBUTION**: 可能性が同じ中で最も近い n だけが選択されます。たとえば、最も近い 20 を選択します。

  ```
  <nearbySelection>
    <blockDistributionSizeMaximum>20</blockDistributionSizeMaximum>
  </nearbySelection>
  ```

- **LINEAR_DISTRIBUTION**: 最も近い要素の中で可能性が高いものを選択します。可能性は線形的に減ります。

  ```
  <nearbySelection>
    <linearDistributionSizeMaximum>40</linearDistributionSizeMaximum>
  </nearbySelection>
  ```

- **PARABOLIC_DISTRIBUTION** (推奨): 最も近い要素の中で、可能性の高いものを選択します。

```
<nearbySelection>
  <parabolicDistributionSizeMaximum>80</parabolicDistributionSizeMaximum>
</nearbySelection>
```

- **BETA_DISTRIBUTION**: ベータディストリビューション に従って選択します。Solver の速度は著しく遅くなります。

```
<nearbySelection>
  <betaDistributionAlpha>1</betaDistributionAlpha>
  <betaDistributionBeta>5</betaDistributionBeta>
</nearbySelection>
```

必要であれば、いつでも ベンチマーカー を使用して値を調整します。

7.7. カスタムの MOVE

7.7.1. 実装に必要な Move のタイプ

お使いの実装に不足している Move のタイプを判断するには、ベンチマーカー を 短期間 実装し、ディスクに最適解を書き込むように設定 します。この最適解を見てください。おそらく局所最適条件になるでしょう。局所最適条件からより早く抜け出せる Move があるかどうか探してみてください。

見つけたら、粒度の粗い Move を実装し、既存の Move と混ぜて、以前の設定に対してベンチマークを設定して評価し、維持するかどうかを確認します。

7.7.2. カスタム Move の導入

一般の **Move** (**ChangeMove** など) を再利用する代わりに、カスタムの **Move** を実装できます。汎用およびカスタムの **MoveSelectors** は、自由に組み合わせることができます。

カスタムの **Move** は、制約を使用して調整できます。たとえば、試験の時間割で、試験 A の時間帯を変更すると、試験 A と同時に行われるすべての試験の時間帯が変更になります。

カスタム **Move** は一般の **Move** よりもわずかに速くなります。ただし、実装に必要な作業ははるかに多く、バグを回避するのもはるかに難しくなります。カスタムの **Move** を実装したら、**environmentMode FULL_ASSERT** を有効にして、スコアが壊れていないか確認します。

7.7.3. Move インターフェース

カスタムの Move には、**Move** インターフェースが実装されている必要があります。

```
public interface Move {

    boolean isMoveDoable(ScoreDirector scoreDirector);

    Move createUndoMove(ScoreDirector scoreDirector);
    void doMove(ScoreDirector scoreDirector);

    Collection<? extends Object> getPlanningEntities();
```

```
    Collection<? extends Object> getPlanningValues();

}
```

クイーン 4 個の **Move** 実装を見てみましょう。クイーンを別の行に移動します。

```
public class RowChangeMove extends AbstractMove {

    private Queen queen;
    private Row toRow;

    public RowChangeMove(Queen queen, Row toRow) {
        this.queen = queen;
        this.toRow = toRow;
    }

    // ... see below

}
```

RowChangeMove のインスタンスは、クイーンを、現在の行から別の行に移動します。

Planner は、**doMove(ScoreDirector)** メソッドを呼び出して Move を行い、それが **doMoveOnGenuineVariables(ScoreDirector)** を呼び出します。**Move** 実装は、**ScoreDirector** に、プランニングエンティティーの変数に行った変更を通知する必要があります。

```
    public void doMoveOnGenuineVariables(ScoreDirector scoreDirector) {
        scoreDirector.beforeVariableChanged(queen, "row"); // before changes are made to the
queen.row
        queen.setRow(toRow);
        scoreDirector.afterVariableChanged(queen, "row"); // after changes are made to the queen.row
    }
```

エンティティーの修正前後に、**scoreDirector.beforeVariableChanged(Object, String)** メソッドと **scoreDirector.afterVariableChanged(Object, String)** メソッドをそれぞれ直接呼び出す必要があります。

注記

1 つの Move で複数のエンティティーを変更して、大きな Move (粒度の粗い Move とも呼ばれています) を効果的に作成できます。

警告

Move は、プランニングエンティティーの変更、追加、または削除だけが可能です。問題ファクトを変更することはできません。

Planner は、Move で **isMoveDoable(ScoreDirector)** メソッドを呼び出し、実行できない Move にフィルターを自動的に設定して排除することができます。実行できない Move は、以下のようになります。

- 現在のソリューションで何も変更しない Move。たとえば、B0 のクイーンはすでに行 0 にいるため、行 0 に移動することはできません。
- 現在のソリューションで実行できない Move。たとえば、B0 のクイーンを行 10 に移動すると境界を越えてしまうため、行 10 に移動することはできません。

N クィーンの例では、現在の行と同じ行にクイーンを動かすことはできません。

```
public boolean isMoveDoable(ScoreDirector scoreDirector) {
    return !ObjectUtils.equals(queen.getRow(), toRow);
}
```

境界の外にクイーンを動かす Move は生成しないため、それを確認する必要はありません。現在実行できない Move は、後のステップで使用している ソリューション を実行可能にできます。

それぞれの Move には 取り消しの **Move** があります。正反対のことを行う (通常は同じタイプの) Move です。上の例では、C0 から C2 の Move に対する取り消しの Move は C2 から C0 です。取り消しの Move は、現在のソリューションで **Move** が行われる前に、**Move** から作成されます。

```
public Move createUndoMove(ScoreDirector scoreDirector) {
    return new RowChangeMove(queen, queen.getRow());
}
```

C0 がすでに C2 に移動してしまったら、取り消しの Move は、C2 から C0 ではなく、C2 から C2 への移動が作成されます。

Solver フェーズは、同じ **Move** の実行とその取り消しを複数回行う可能性があります。つまり、Solver フェーズの多くは、Move を評価するために何度も Move の実行と取り消しを行ってから、そのうちの 1 つを選択して Move を行います (この時は取り消しは行われません)。

Move には、**getPlanningEntities()** メソッドおよび **getPlanningValues()** メソッドが実装されている必要があります。このメソッドは、エンティティーのタプーと値のタプーにそれぞれ使用されます。**Move** が行われると、このメソッドが呼び出されます。

```
public List<? extends Object> getPlanningEntities() {
    return Collections.singletonList(queen);
}

public Collection<? extends Object> getPlanningValues() {
    return Collections.singletonList(toRow);
}
```

Move が複数のプランニングエンティティーを変更した場合は、**getPlanningEntities()** でそのすべてを返し、**getPlanningValues()** で、そのすべての値を (変更しているものに) 返します。

```
public Collection<? extends Object> getPlanningEntities() {
    return Arrays.asList(leftCloudProcess, rightCloudProcess);
}

public Collection<? extends Object> getPlanningValues() {
    return Arrays.asList(leftCloudProcess.getComputer(), rightCloudProcess.getComputer());
}
```

Move は、**equals()** メソッドと **hashCode()** メソッドを実装する必要があります。ソリューションで同じ変更を作成する 2 つの Move は同等である必要があります。

```
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        } else if (o instanceof RowChangeMove) {
            RowChangeMove other = (RowChangeMove) o;
            return new EqualsBuilder()
                    .append(queen, other.queen)
                    .append(toRow, other.toRow)
                    .isEquals();
        } else {
            return false;
        }
    }

    public int hashCode() {
        return new HashCodeBuilder()
                .append(queen)
                .append(toRow)
                .toHashCode();
    }
```

これは、別の Move が、同じ Move タイプのインスタンスかどうかを確認していることに注意してください。この **instanceof** チェックは、1つ以上の Move タイプを使用している場合に、Move を別の Move タイプと比較するため、重要になります。

toString() メソッドを実装して、Planner のログが簡単に読めるようにします。

```
    public String toString() {
        return queen + " {" + queen.getRow() + " -> " + toRow + "}";
    }
```

カスタムの **Move** を1つだけ実装できます。このようなカスタムの Move を生成してみましょう。

7.7.4. MoveListFactory: カスタム Move を簡単に生成する方法

カスタム Move を生成する一番簡単な方法は、**MoveListFactory** インターフェースを実装することです。

```
public interface MoveListFactory<S extends Solution> {

    List<Move> createMoveList(S solution);

}
```

例:

```
public class RowChangeMoveFactory implements MoveListFactory<NQueens> {

    public List<Move> createMoveList(NQueens nQueens) {
        List<Move> moveList = new ArrayList<Move>();
        for (Queen queen : nQueens.getQueenList()) {
            for (Row toRow : nQueens.getRowList()) {
                moveList.add(new RowChangeMove(queen, toRow));
            }
```

```
        }
        return moveList;
    }

}
```

簡単な設定方法 (その他の **MoveSelector** と同様、**unionMoveSelector** にネストできます):

```
    <moveListFactory>

<moveListFactoryClass>org.optaplanner.examples.nqueens.solver.move.factory.RowChangeMoveFact
ory</moveListFactoryClass>
    </moveListFactory>
```

高度な設定方法:

```
    <moveListFactory>
      ... <!-- Normal moveSelector properties -->

<moveListFactoryClass>org.optaplanner.examples.nqueens.solver.move.factory.RowChangeMoveFact
ory</moveListFactoryClass>
    </moveListFactory>
```

MoveListFactory は、**List<Move>** ですべての Move を一度に生成するため、**cacheType** の **JUST_IN_TIME** には対応しません。したがって、**moveListFactory** は **cacheType STEP** をデフォルトで使用するため、メモリーフットプリントが著しく増えます。

7.7.5. MoveIteratorFactory: カスタム Move の Just in Time を生成

この高度な構造を使用して、**MoveIteratorFactory** インターフェースを実装して、カスタム Move を生成します。

```
public interface MoveIteratorFactory {

    long getSize(ScoreDirector scoreDirector);

    Iterator<Move> createOriginalMoveIterator(ScoreDirector scoreDirector);

    Iterator<Move> createRandomMoveIterator(ScoreDirector scoreDirector, Random
workingRandom);

}
```

getSize() メソッドは、サイズの概算を取得します。正確である必要はありません。**selectionOrder** が **ORIGINAL** の場合、もしくはキャッシュされる場合は、**createOriginalMoveIterator** メソッドが呼び出されます。**selectionOrder RANDOM** が **cacheType JUST_IN_TIME** とともに使用されている場合は、**createRandomMoveIterator** メソッドが呼び出されます。

重要

Iterator<Move> の作成時には、**Move** コレクション (リスト、アレイ、マップ、セット) は作成されません。**MoveListFactory** における **MoveIteratorFactory** の一番の目的は、**Iterator** の **next()** メソッドの **Move** ジャストインタイムを作成できるようにすることです。

簡単な設定方法 (その他の **MoveSelector** と同様、**unionMoveSelector** にネストできます):

```
<moveIteratorFactory>
  <moveIteratorFactoryClass>...</moveIteratorFactoryClass>
</moveIteratorFactory>
```

高度な設定方法:

```
<moveIteratorFactory>
  ... <!-- Normal moveSelector properties -->
  <moveIteratorFactoryClass>...</moveIteratorFactoryClass>
</moveIteratorFactory>
```

第8章 EXHAUSTIVE SEARCH (しらみつぶし探索)

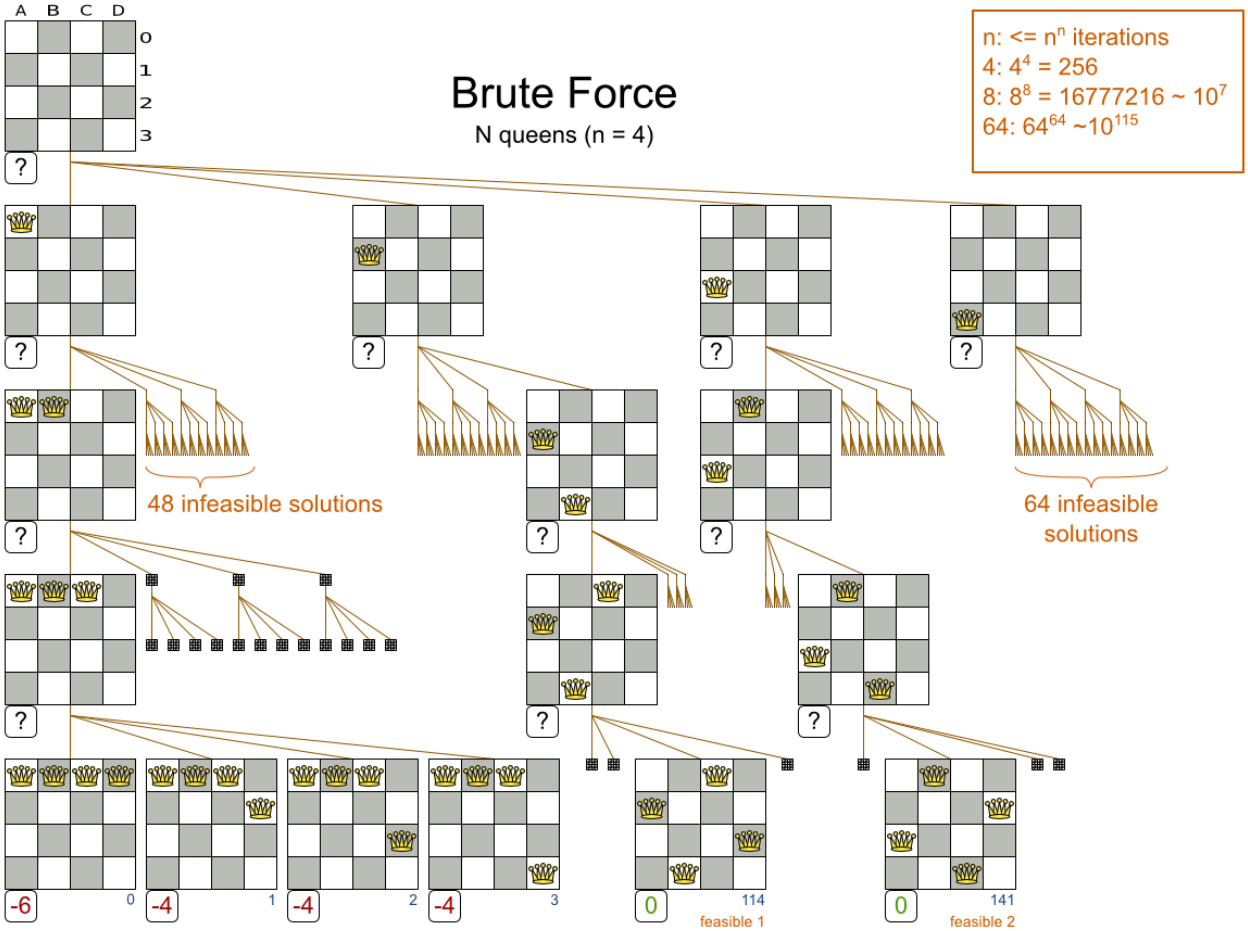
8.1. 概要

しらみつぶし探索は常に大域的最適を見つけ認識しますが、スケールは行わない (その小規模のデータセットを超えることはない) ため、大抵は役に立ちません。

8.2. BRUTE FORCE (力まかせ)

8.2.1. アルゴリズムの説明

力まかせアルゴリズムは、可能性のあるすべての解を作成して評価します。



問題のサイズが大きくなると、探索ツリーが指数関数的に大きくなるため、スケーラビリティーの問題が発生します。

重要

しらみつぶし探索のスケーラビリティー で説明しているように、力まかせは、通常、時間に制約のある現実社会で使用するには適しません。

8.2.2. 設定

力まかせの最も簡単な設定方法:

```
<solver>
  ...
  <exhaustiveSearch>
    <exhaustiveSearchType>BRUTE_FORCE</exhaustiveSearchType>
  </exhaustiveSearch>
</solver>
```

8.3. BRANCH AND BOUND (分枝限定)

8.3.1. アルゴリズムの説明

分岐限定も、指数関数的な探索木でノードを検証しますが、より見込みのあるノードを最初に調査して、見込みのないノードを取り除きます。

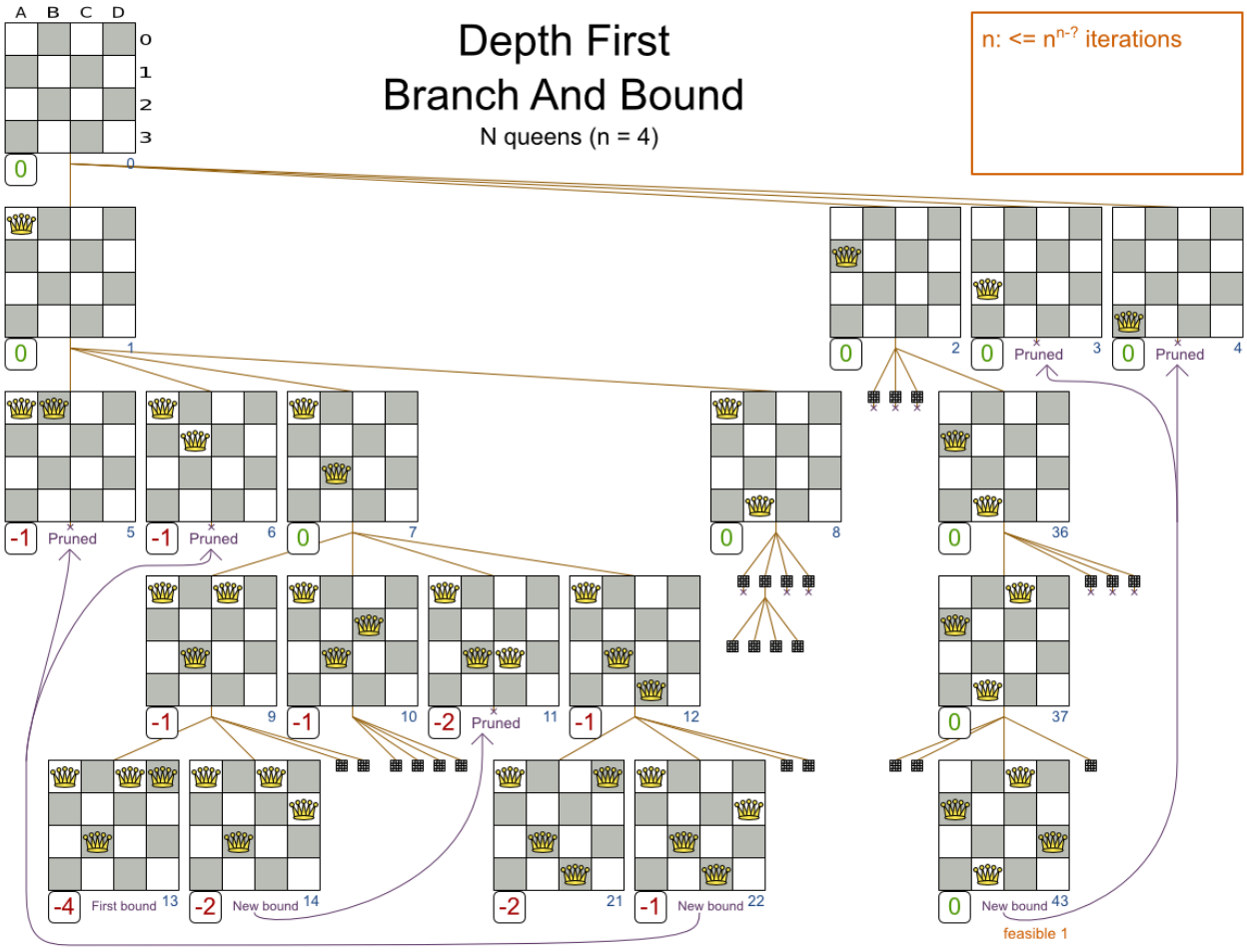
各ノードに対して、分岐限定は、最適な限界 (そのノードがつながる最適な可能スコア) を計算します。ノードの最適な限界が、グローバルの悲観的な限界と同じか、低くなる場合、そのノードは取り除かれます (その全サブノードの分岐全体を含む)。

注記

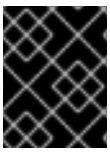
学術論文では、楽観限界 (optimistic bound) の代わりに下界 (lower bound) という用語を使用し、悲観限界 (pessimistic bound) の代わりに上界 (upper bound) という用語を使用します。これは、スコアを最小限に抑えるためです。

Planner は、(否定または肯定の制約を組み合わせて対応するため) スコアを最大限に活用します。したがって、それを明確にするために異なる用語を使用します。常に高い限界なのに下限という用語を使用すると紛らわしくなるためです。

たとえば、インデックス 14 で、グローバルの楽観限界を **-2** に設定します。インデックス 11 にたどり着いたノードから到達できるすべての解が、**-2** (ノードの楽観限界)と同じか、低くなるため、その解は除外できます。



(力まかせ とほぼ同じく) 分岐限定は、問題のサイズが大きくなると指数関数的に大きくなる探索木を作成します。したがって、少し遅れるだけで、結局は同じスケーラビリティーの壁に到達します。



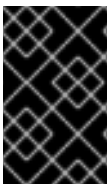
重要

分岐限定は、時間が限られているため、通常は、しらみつぶし探索のスケーラビリティー で説明しているように、現実の問題に対してはほぼ使用できません。

8.3.2. 設定

分枝限定の最も簡単な設定:

```
<solver>
  ...
  <exhaustiveSearch>
    <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
  </exhaustiveSearch>
</solver>
```



重要

枝刈りが、デフォルトの **ScoreBounder** と連携するには、InitializingScoreTrend を設定する必要があります。特に、**ONLY_DOWN** の InitializingScoreTrend (または最高スコアレベルに **ONLY_DOWN** がある) が、多くを除外します。

高度な設定方法:

```
<exhaustiveSearch>
  <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
  <nodeExplorationType>DEPTH_FIRST</nodeExplorationType>
  <entitySorterManner>DECREASING_DIFFICULTY_IF_AVAILABLE</entitySorterManner>
  <valueSorterManner>INCREASING_STRENGTH_IF_AVAILABLE</valueSorterManner>
</exhaustiveSearch>
```

nodeExplorationType オプション:

- **DEPTH_FIRST** (デフォルト): より深いノードを最初に調べます (そしてスコアが良いもの、より楽観的な限界の順)。より深いノード (特に葉節点) は悲観限界だけを改善します。悲観限界を向上させると、ノードがさらに除外されるため、探索領域が減ります。

  ```
  <exhaustiveSearch>
    <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
    <nodeExplorationType>DEPTH_FIRST</nodeExplorationType>
  </exhaustiveSearch>
  ```

- **BREADTH_FIRST** (推奨されません): 1 レイヤーずつ (次にスコアが良いもの、楽観限界が良いものの順に)、ノードを調べます。メモリーで (そして通常はパフォーマンスでも) 大きくスケールします。

  ```
  <exhaustiveSearch>
    <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
    <nodeExplorationType>BREADTH_FIRST</nodeExplorationType>
  </exhaustiveSearch>
  ```

- **SCORE_FIRST**: スコアが高いノードを最初に調べます (そして、より楽観的な限界、ノードが深いものの順に)。場合によっては、**BREADTH_FIRST** と同じだけスケールします。

  ```
  <exhaustiveSearch>
    <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
    <nodeExplorationType>SCORE_FIRST</nodeExplorationType>
  </exhaustiveSearch>
  ```

- **OPTIMISTIC_BOUND_FIRST**: 楽観的限界が良いノードを最初に調べます (そして、スコアが良いもの、ノードが深いものの順)。場合によっては、**BREADTH_FIRST** と同じだけスケールします。

  ```
  <exhaustiveSearch>
    <exhaustiveSearchType>BRANCH_AND_BOUND</exhaustiveSearchType>
    <nodeExplorationType>OPTIMISTIC_BOUND_FIRST</nodeExplorationType>
  </exhaustiveSearch>
  ```

entitySorterManner オプションは、以下のようになります。

- **DECREASING_DIFFICULTY**: 難しいプランニングエンティティーが最初に初期化されます。これにより、通常は、枝刈りが改善 (したがってスケーラビリティーが改善) します。モデルが プランニングエンティティーの難易度比較 に対応している必要があります。

- **DECREASING_DIFFICULTY_IF_AVAILABLE** (デフォルト): モデルが プランニングエンティティーの難易度比較 に対応してる場合は **DECREASING_DIFFICULTY** と同じ動作になり、対応していない場合は **NONE** と同じになります。

- **NONE**: 元の順番で、プランニングエンティティーを初期化します。

valueSorterManner オプションは、以下のようになります。

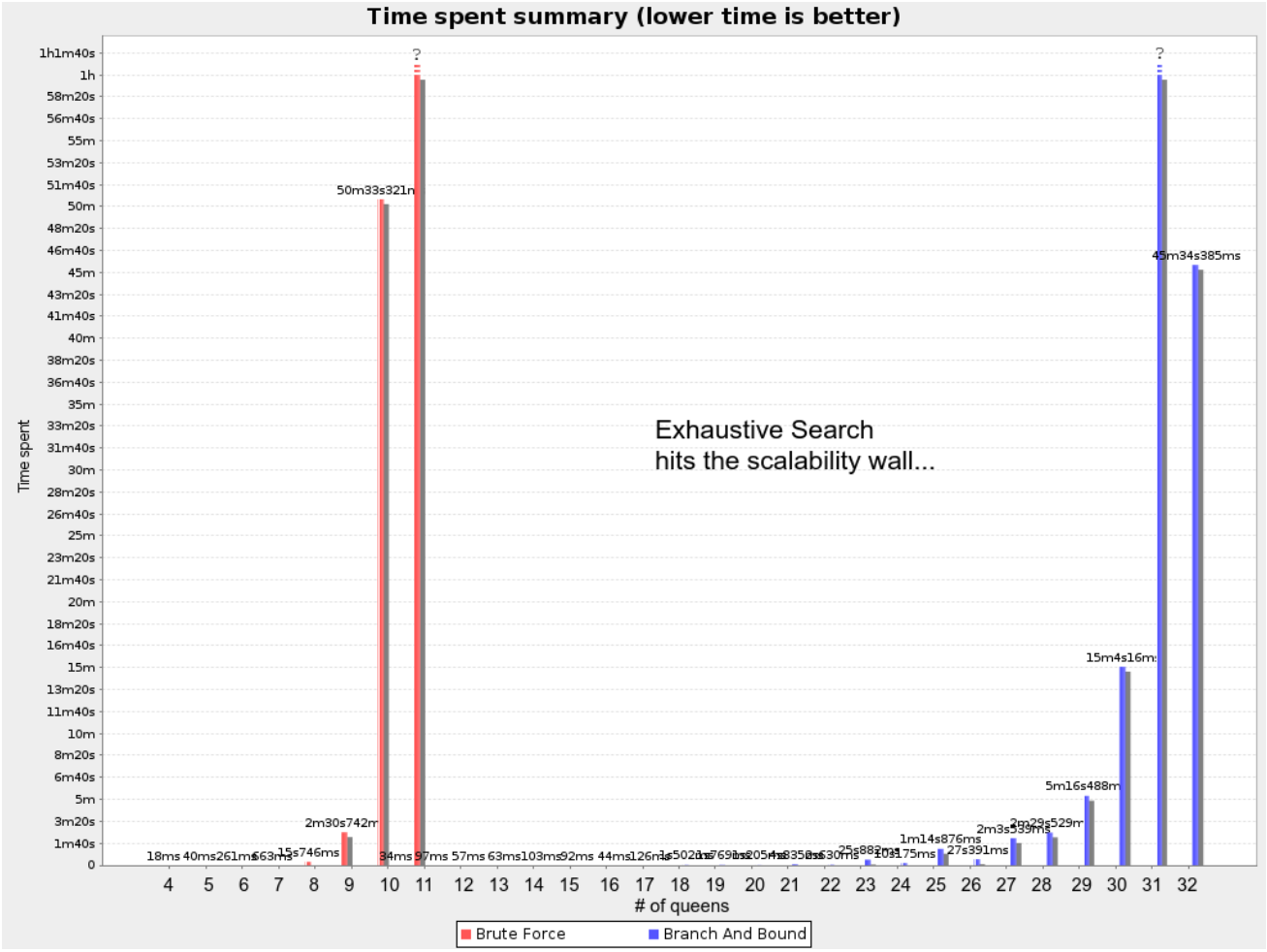
- **INCREASING_STRENGTH**: 計画値の強度が小さいものから順に評価します。モデルが 計画値の強度比較 に対応している必要があります。
- **INCREASING_STRENGTH_IF_AVAILABLE** (デフォルト): モデルが 計画値の強度比較 に対応している場合は **INCREASING_STRENGTH** と同じ動作になり、対応していない場合は **NONE** と同じになります。
- **DECREASING_STRENGTH**: 計画値の強度が大きいものから順に評価します。モデルが 計画値の強度比較 に対応している必要があります。
- **DECREASING_STRENGTH_IF_AVAILABLE**: モデルが 計画値の強度比較 に対応している場合は **DECREASING_STRENGTH** と同じ動作になります。対応していない場合は **NONE** と同じになります。
- **NONE**: 元の順番で、計画値を試します。

8.4. しらみつぶし探索のスケーラビリティー

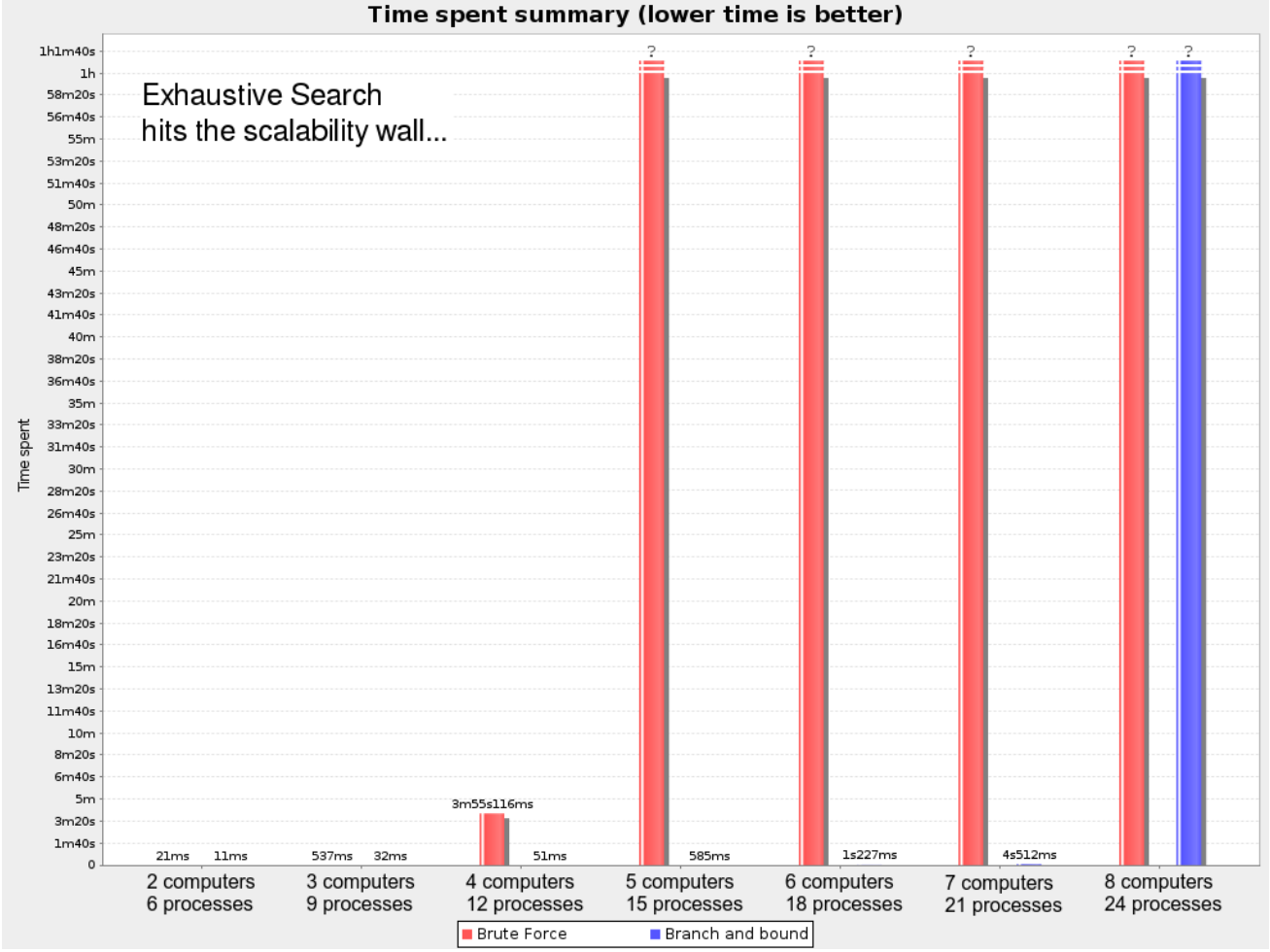
各種しらみつぶし探索には、以下のような重要なスケーラビリティー問題があります。

- メモリー使用のスケーラビリティーが極めて低い。
- パフォーマンスのスケーラビリティーが極めて低い。

ベンチマーカー における、時間経過のグラフで示されるように、力まかせ (Brute Force) と分岐限定 (Branch And Bound) の両方に、パフォーマンスのスケーラビリティーの壁が立ちはだかります。たとえば、N クィーンでは、クィーンの数が数十になれば行き詰まります。



クラウドのバランスなどの多くのユースケースでは、壁が突然出現します。



しらみつぶし探索は、データセットが小さい場合でもすでにこの壁があります。したがって、本番環境では、この最適化アルゴリズムはほとんど役に立ちません。代わりに構築ヒューリスティックと局所探索法を組み合わせ使用します。これにより、クイーンやコンピューターの数が数千になっても、簡単に処理できます。

注記

このスケーラビリティー問題について、ハードウェアでは目立った影響はありません。新しいハードウェアが出現しても、大海の中のひとしずくとなります。ムーアの法則ですら、データセットにさらにいくつかのプランニングエンティティーが発生すれば勝てません。

第9章 CONSTRUCTION HEURISTICS (構築ヒューリスティック)

9.1. 概要

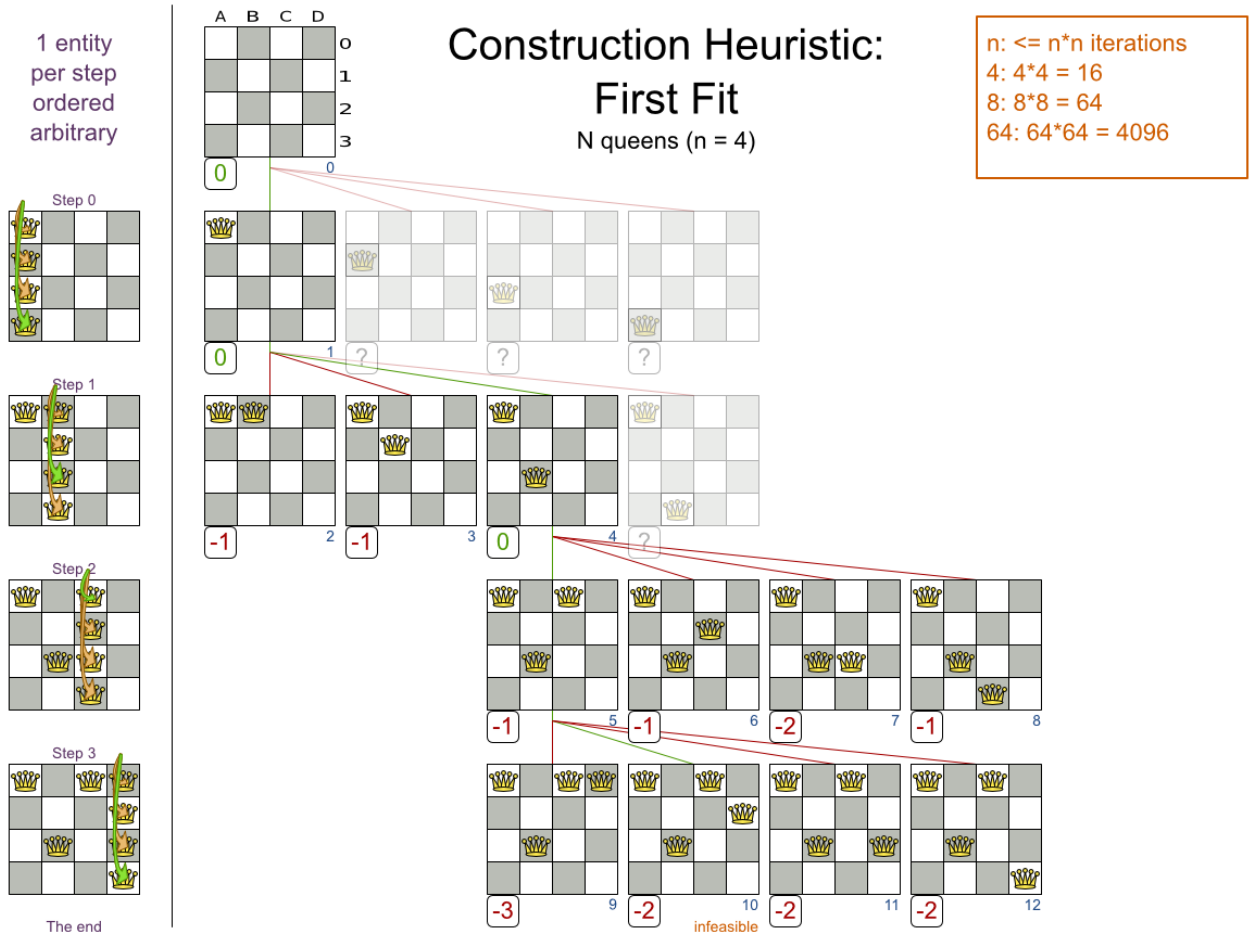
構築ヒューリスティックは、時間が限られている中でかなり良い初期解を構築します。構築された解が常に実行可能とは限りませんが、すぐに見つけることができるため、メタヒューリスティックがジョブを終了できます。

構築ヒューリスティックは自動的に終了するため、構築ヒューリスティックフェーズに **Termination** を明示的に設定する必要はありません。

9.2. FIRST FIT (FF)

9.2.1. アルゴリズムの説明

FF (First Fit) アルゴリズムは、すべてのプランニングエンティティーを (デフォルトの順番で) 繰り返します。一度に初期化するプランニングエンティティーは1つです。これは、プランニングエンティティーを、利用できる最適なプランニングエンティティーに割り当て、すでに初期化されているプランニングエンティティーを考慮します。プランニングエンティティーがすべて初期化されると終了します。プランニングエンティティーを割り当てたら変更しません。



これは、クイーン A を行 0 で開始 (そしてその後は移動しない) していることに注意してください。これが最適解に到達することはできません。メタヒューリスティックを使用してこの構築ヒューリスティックにサフィックスを追加すると、これが改善できます。

9.2.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT</constructionHeuristicType>
</constructionHeuristic>
```

注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

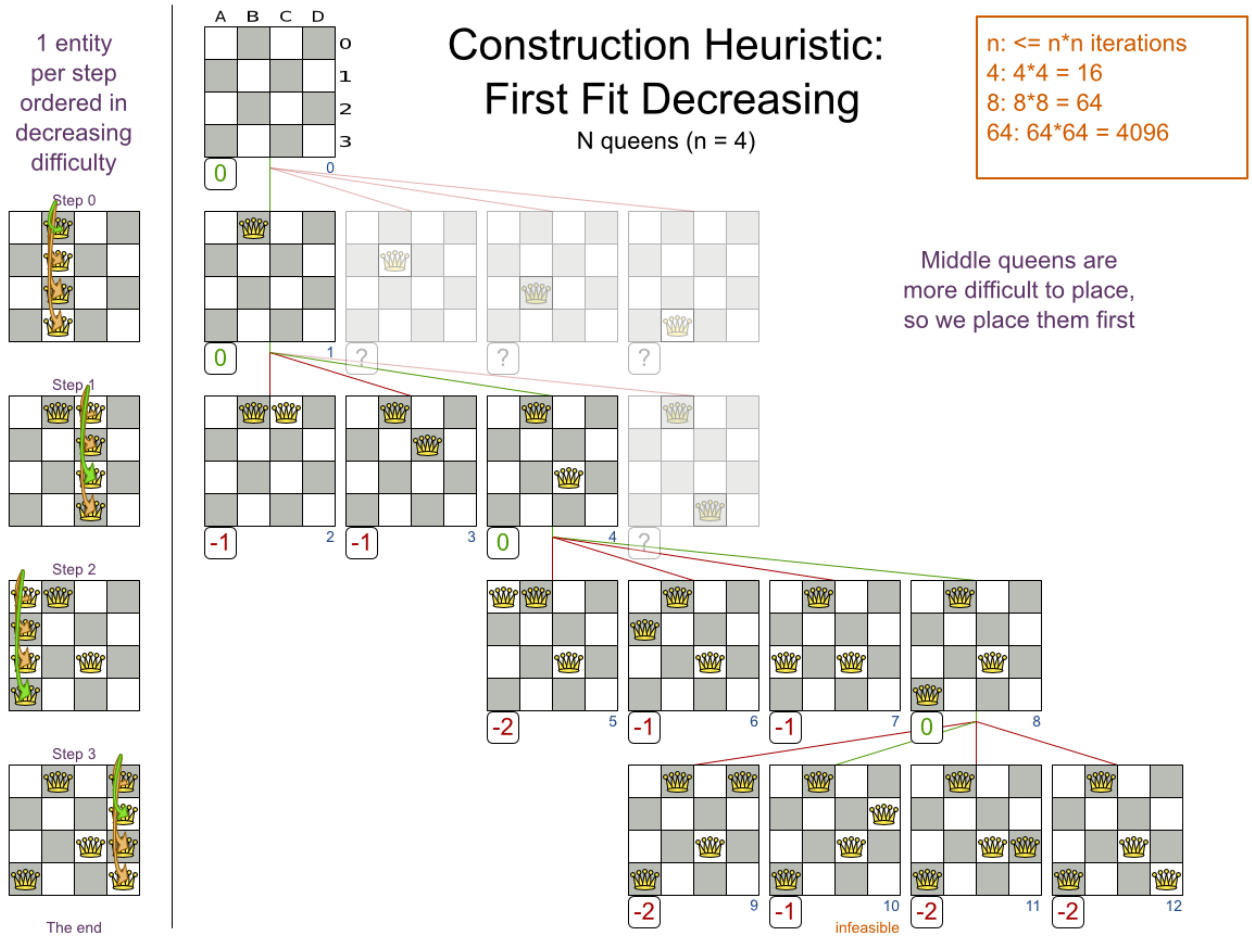
高度な設定については、キューからのエンティティーの割り当て を参照してください。

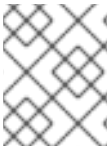
9.3. FIRST FIT DECREASING (FFD)

9.3.1. アルゴリズムの説明

FF (First Fit) と同じですが、より難しいプランニングエンティティーを最初に割り当てます。おそらく、残ったものが適合する可能性はないためです。したがって、難易度順にプランニングエンティティーをソートします。

モデルが プランニングエンティティーの難易度比較 に対応している必要があります。





注記

このアルゴリズムは、通常、FF (First Fit) よりも良い結果が得られると期待される場合もありますが、常に得られるとは限りません。

9.3.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```

注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

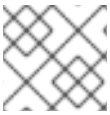
高度な設定については、キューからのエンティティーの割り当て を参照してください。

9.4. WEAKEST FIT (WF)

9.4.1. アルゴリズムの説明

FF (First Fit) と同じですが、計画値が弱いものを最初に使用します。計画値が高い方が、あとでプランニングエンティティーを提供する可能性が高いためです。したがって、弱いものから順に計画値をソートします。

モデルが プランニングエンティティーの強度比較 に対応している必要があります。



注記

このアルゴリズムが FF (First Fit) よりも良い結果が得られることはあまりありません。

9.4.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>WEAKEST_FIT</constructionHeuristicType>
</constructionHeuristic>
```

注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

高度な設定については、キューからのエンティティーの割り当て を参照してください。

9.5. WEAKEST FIT DECREASING (WFD)

9.5.1. アルゴリズムの説明

FFD (First Fit Decreasing) と WF (Weakest Fit) を組み合わせています。したがって、難易度が高く、強度が低いものからプランニングエンティティーをソートします。

モデルが、プランニングエンティティーの難易度比較 および 計画値の強度比較 に対応している必要があります。

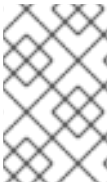
注記

このアルゴリズムが、FFD (First Fit Decreasing) よりも良い結果を得られることはほとんどありませんが、通常は WF (Weakest Fit) よりも良い結果となります。

9.5.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>WEAKEST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```



注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

高度な設定については、キューからのエンティティーの割り当て を参照してください。

9.6. STRONGEST FIT (SF)

9.6.1. アルゴリズムの説明

FF (First Fit) と同じですが、計画値が強いものを最初に使用します。計画値が強い方が、おそらく使用するソフトコストが低いためです。したがって、強度が高いものから順に計画値をソートします。

モデルが プランニングエンティティーの強度比較 に対応している必要があります。

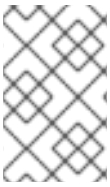
注記

このアルゴリズムが、FF (First Fit) または WF (Weakest Fit) よりも良い結果を得られることはあまりありません。

9.6.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>STRONGEST_FIT</constructionHeuristicType>
</constructionHeuristic>
```



注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

高度な設定については、キューからのエンティティーの割り当て を参照してください。

9.7. STRONGEST FIT DECREASING (SFD)

9.7.1. アルゴリズムの説明

FFD (First Fit Decreasing) と SF (Strongest Fit) を組み合わせます。したがって、プランニングエンティティーの難易度が高く、計画値の強度が高い順にソートします。

モデルが、プランニングエンティティーの難易度比較 および 計画値の強度比較 に対応している必要があります。

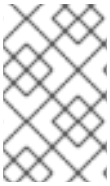
注記

このアルゴリズムが、FFD (First Fit Decreasing) または WFD (Weakest Fit Decreasing) よりも良い結果を得られることはほとんどありませんが、通常は SF (Strongest Fit) よりも良い結果となります。

9.7.2. 設定

次の Solver フェーズを設定します。

```
<constructionHeuristic>
  <constructionHeuristicType>STRONGEST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```



注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

高度な設定については、キューからのエンティティーの割り当て を参照してください。

9.8. ALLOCATE ENTITY FROM QUEUE (AEFQ)

9.8.1. アルゴリズムの説明

AEFQ (Allocate Entity From Queue) は、FF (First Fit)、FFD (First Fit Decreasing)、WF (Weakest Fit) および WFD (Weakest Fit Decreasing) に関する、用途が広い、一般的な形式です。以下のように振舞います。

1. すべてのエンティティーをキューに追加する。
2. (そのキューの) 最初のエンティティーを、最適値に割り当てる。
3. すべてのエンティティーが割り当てられるまで繰り返す。

9.8.2. 設定

簡単な設定方法:

```
<constructionHeuristic>
  <constructionHeuristicType>ALLOCATE_ENTITY_FROM_QUEUE</constructionHeuristicType>
</constructionHeuristic>
```

詳細で簡単な設定方法:

```
<constructionHeuristic>
  <constructionHeuristicType>ALLOCATE_ENTITY_FROM_QUEUE</constructionHeuristicType>
  <entitySorterManner>DECREASING_DIFFICULTY_IF_AVAILABLE</entitySorterManner>
  <valueSorterManner>INCREASING_STRENGTH_IF_AVAILABLE</valueSorterManner>
</constructionHeuristic>
```

entitySorterManner オプションは、以下のようになります。

- **DECREASING_DIFFICULTY**: 難しいプランニングエンティティーが最初に初期化されます。これにより、通常は、枝刈りが改善 (したがってスケーラビリティーが改善) します。モデルが プランニングエンティティーの難易度比較 に対応している必要があります。
- **DECREASING_DIFFICULTY_IF_AVAILABLE** (デフォルト): モデルが プランニングエンティティーの難易度比較 に対応してる場合は **DECREASING_DIFFICULTY** と同じ動作になり、対応していない場合は **NONE** と同じになります。
- **NONE**: 元の順番で、プランニングエンティティーを初期化します。

valueSorterManner オプションは、以下のようになります。

- **INCREASING_STRENGTH**: 計画値の強度が小さいものから順に評価します。モデルが 計画値の強度比較 に対応している必要があります。
- **INCREASING_STRENGTH_IF_AVAILABLE** (デフォルト): モデルが 計画値の強度比較 に対応している場合は **INCREASING_STRENGTH** と同じ動作になり、対応していない場合は **NONE** と同じになります。
- **DECREASING_STRENGTH**: 計画値の強度が大きいものから順に評価します。モデルが 計画値の強度比較 に対応している必要があります。
- **DECREASING_STRENGTH_IF_AVAILABLE**: モデルが 計画値の強度比較 に対応している場合は **DECREASING_STRENGTH** と同じ動作になります。対応していない場合は **NONE** と同じになります。
- **NONE**: 元の順番で、計画値を試します。

高度で詳細な設定方法 (たとえば、1 つのエンティティークラスに 1 つの変数がある場合の WFD (Weakest Fit Decreasing) 設定):

```
<constructionHeuristic>
  <queuedEntityPlacer>
    <entitySelector id="placerEntitySelector">
      <cacheType>PHASE</cacheType>
      <selectionOrder>SORTED</selectionOrder>
      <sorterManner>DECREASING_DIFFICULTY</sorterManner>
    </entitySelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="placerEntitySelector"/>
      <valueSelector>
        <cacheType>PHASE</cacheType>
        <selectionOrder>SORTED</selectionOrder>
        <sorterManner>INCREASING_STRENGTH</sorterManner>
      </valueSelector>
    </changeMoveSelector>
  </queuedEntityPlacer>
</constructionHeuristic>
```

ステップごとに、**QueuedEntityPlacer** は、**EntitySelector** から初期化されていないエンティティーを 1 つ選択し、(**MoveSelector** が生成したエンティティーに対するすべての Move から) 勝利 **Move** を適用します。模倣選択 は、勝利 **Move** が、選択したエンティティー (だけ) を変更します。

エンティティーまたは値のソートをカスタマイズするには、ソートした選択 を参照してください。その他の セレクター のカスタマイズ (filtering や limiting など) も対応しています。

9.8.3. 複数の変数

複数の変数を扱う方法が 2 つあります。 **ChangeMove** の組み合わせ方法が異なります。

- **ChangeMove** のデカルト積 (デフォルト): 選択したエンティティーの変数はすべて一緒に割り当てます。(特に時間割ユースケースで) 結果がはるかに改善されます。
- 一連の **ChangeMove**: 一度に 1 つの変数が割り当てられます。特に 3 つ以上の変数がある場合に、スケールがはるかに良くなります。

たとえば、「コースの時間割」の例で、部屋が 200、時間帯が 40 あるとします。

変数が 2 つある 1 つのエンティティークラスの場合、この FF (First Fit) 設定は、**ChangeMove** の デカルト積 を使用して、エンティティーごとに Move を 8000 個選択します。

```
<constructionHeuristic>
  <queuedEntityPlacer>
    <entitySelector id="placerEntitySelector">
      <cacheType>PHASE</cacheType>
    </entitySelector>
    <cartesianProductMoveSelector>
      <changeMoveSelector>
        <entitySelector mimicSelectorRef="placerEntitySelector"/>
        <valueSelector>
          <variableName>room</variableName>
        </valueSelector>
      </changeMoveSelector>
```

```
      <changeMoveSelector>
       <entitySelector mimicSelectorRef="placerEntitySelector"/>
       <valueSelector>
        <variableName>period</variableName>
       </valueSelector>
      </changeMoveSelector>
     </cartesianProductMoveSelector>
    </queuedEntityPlacer>
    ...
</constructionHeuristic>
```

警告

それぞれ 1000 個の値が含まれる変数を 3 つ使用して、デカルト積が、1 つのエンティティーにつき値を 1 000 000 000 個選択するため、時間がかかります。

変数が 2 つある 1 つのエンティティークラスに対するこの FF (First Fit) 設定は、 順次的な **ChangeMove** を使用して、エンティティーごとに Move を 240 個選択します。

```
<constructionHeuristic>
  <queuedEntityPlacer>
    <entitySelector id="placerEntitySelector">
      <cacheType>PHASE</cacheType>
    </entitySelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="placerEntitySelector"/>
      <valueSelector>
        <variableName>period</variableName>
      </valueSelector>
    </changeMoveSelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="placerEntitySelector"/>
      <valueSelector>
        <variableName>room</variableName>
      </valueSelector>
    </changeMoveSelector>
  </queuedEntityPlacer>
  ...
</constructionHeuristic>
```

重要

特に順次的な **ChangeMove** の場合は、変数の順番は重要です。上記の例では、時間帯を、後ではなく先に選択することが推奨されます。講義室には関係ないハード制約が多くあるためです (たとえば、教師が授業を同時に 2 つ受け持つことはできない)。ベンチマーカー が指針となります。

変数が 3 つ以上ある場合は、デカルト積と逐次方式を組み合わせることができます。

```
<constructionHeuristic>
  <queuedEntityPlacer>
    ...
    <cartesianProductMoveSelector>
      <changeMoveSelector>...</changeMoveSelector>
      <changeMoveSelector>...</changeMoveSelector>
    </cartesianProductMoveSelector>
    <changeMoveSelector>...</changeMoveSelector>
  </queuedEntityPlacer>
  ...
</constructionHeuristic>
```

9.8.4. 複数のエンティティークラス

複数のエンティティークラスを扱う最も簡単な方法は、各エンティティークラスに対して異なる構築ヒューリスティックを実行することです。

```
<constructionHeuristic>
  <queuedEntityPlacer>
    <entitySelector id="placerEntitySelector">
      <cacheType>PHASE</cacheType>
      <entityClass>...DogEntity</entityClass>
    </entitySelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="placerEntitySelector"/>
    </changeMoveSelector>
  </queuedEntityPlacer>
  ...
</constructionHeuristic>
<constructionHeuristic>
  <queuedEntityPlacer>
    <entitySelector id="placerEntitySelector">
      <cacheType>PHASE</cacheType>
      <entityClass>...CatEntity</entityClass>
    </entitySelector>
    <changeMoveSelector>
      <entitySelector mimicSelectorRef="placerEntitySelector"/>
    </changeMoveSelector>
  </queuedEntityPlacer>
  ...
</constructionHeuristic>
```

9.8.5. 早期発見のタイプ

構築ヒューリスティックには、早期発見のタイプがいくつかあります。

- **NEVER**: 変数を初期化するのに選択した Move をすべて評価します。これは、InitializingScoreTrend が **ONLY_DOWN** に設定されていない場合のデフォルトです。

```
<constructionHeuristic>
  ...
  <forager>
```

```
    <pickEarlyType>NEVER</pickEarlyType>
  </forager>
</constructionHeuristic>
```

- **FIRST_NON_DETERIORATING_SCORE**: スコアを下げない最初の Move で変数を初期化します。選択した残りの Move は無視します。これは、InitializingScoreTrend が **ONLY_DOWN** である場合のデフォルトです。

```
<constructionHeuristic>
  ...
  <forager>
    <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType>
  </forager>
</constructionHeuristic>
```

注記

否定的な制約だけがあり、厳密に言うと InitializingScoreTrend は **ONLY_DOWN** ではない場合に、FIRST_NON_DETERIORATING_SCORE を適用するのが理にかなっている可能性があります。スコアの品質が失われると時間が短くなるかどうかを判断するには ベンチマーカー を使用します。

- **FIRST_FEASIBLE_SCORE**: 適切なスコアがある最初の Move で変数を初期化します。

```
<constructionHeuristic>
  ...
  <forager>
    <pickEarlyType>FIRST_FEASIBLE_SCORE</pickEarlyType>
  </forager>
</constructionHeuristic>
```

InitializingScoreTrend が **ONLY_DOWN** の場合は、**FIRST_FEASIBLE_SCORE_OR_NON_DETERIORATING_HARD** を代わりに使用してください。この方が速く、デメリットもありません。

- **FIRST_FEASIBLE_SCORE_OR_NON_DETERIORATING_HARD**: スコアの実現可能性を低下しない最初の Move で変数を初期化します。

```
<constructionHeuristic>
  ...
  <forager>

<pickEarlyType>FIRST_FEASIBLE_SCORE_OR_NON_DETERIORATING_HARD</pickEarlyType>
  </forager>
</constructionHeuristic>
```

9.9. ALLOCATE TO VALUE FROM QUEUE (ATVFQ)

9.9.1. アルゴリズムの説明

ATVFQ (Allocate To Value From Queue) は、以下のように働きます。

1. すべての値をラウンドロビンキューにおきます。
2. (キューから) 最初の値に最適なエンティティーを割り当てます。
3. すべてのエンティティーが割り当てられるまで繰り返す。

9.9.2. 設定

簡単な設定方法:

```
<constructionHeuristic>

<constructionHeuristicType>ALLOCATE_TO_VALUE_FROM_QUEUE</constructionHeuristicType>
</constructionHeuristic>
```

詳細で簡単な設定方法:

```
<constructionHeuristic>

<constructionHeuristicType>ALLOCATE_TO_VALUE_FROM_QUEUE</constructionHeuristicType>
  <entitySorterManner>DECREASING_DIFFICULTY_IF_AVAILABLE</entitySorterManner>
  <valueSorterManner>INCREASING_STRENGTH_IF_AVAILABLE</valueSorterManner>
</constructionHeuristic>
```

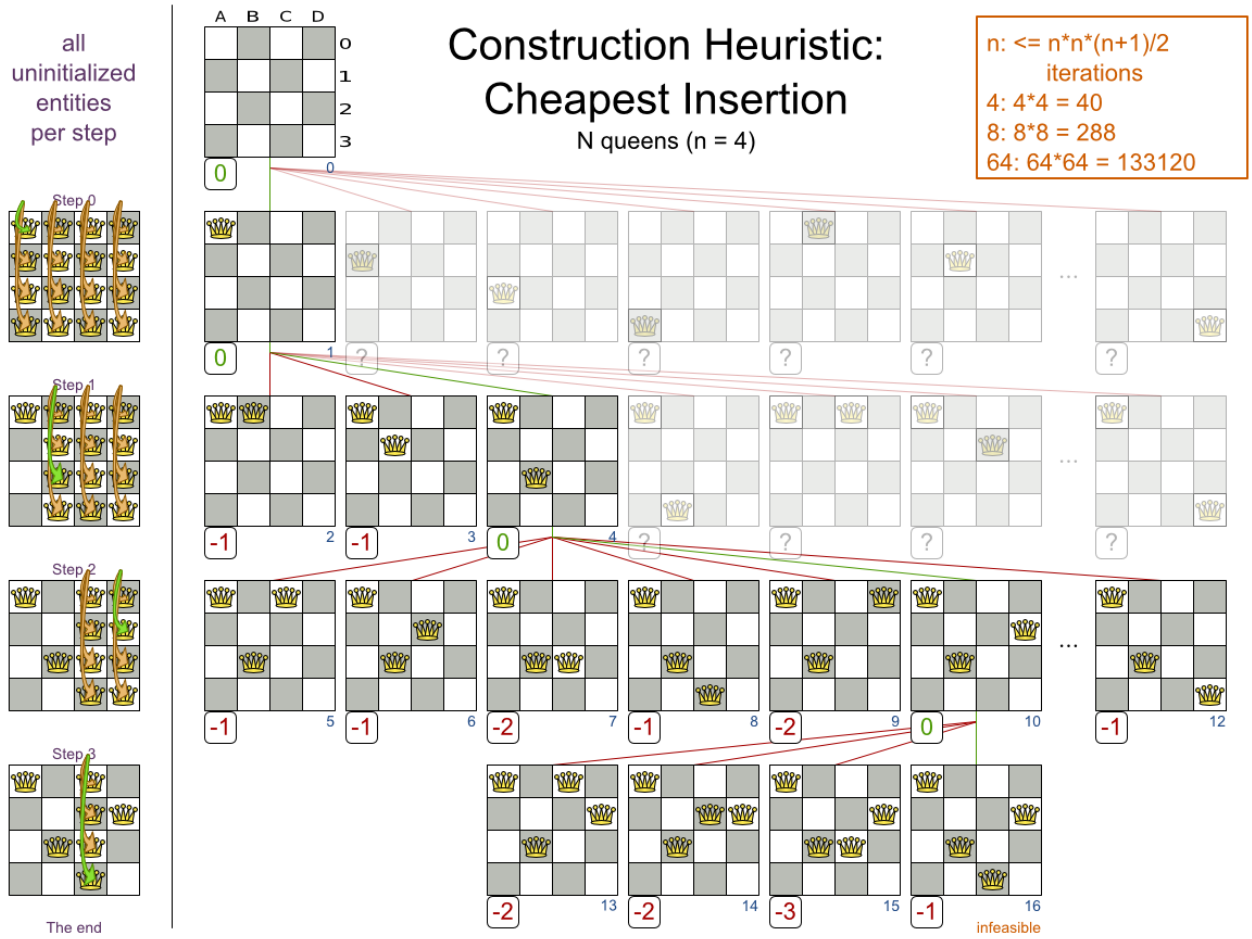
高度で詳細な設定方法: (たとえば、変数が 1 つある 1 つのエンティティークラスへの設定)

```
<constructionHeuristic>
  <queuedValuePlacer>
    <valueSelector id="placerValueSelector">
      <cacheType>PHASE</cacheType>
      <selectionOrder>SORTED</selectionOrder>
      <sorterManner>INCREASING_STRENGTH</sorterManner>
    </valueSelector>
    <changeMoveSelector>
      <entitySelector>
        <cacheType>PHASE</cacheType>
        <selectionOrder>SORTED</selectionOrder>
        <sorterManner>DECREASING_DIFFICULTY</sorterManner>
      </entitySelector>
      <valueSelector mimicSelectorRef="placerValueSelector"/>
    </changeMoveSelector>
  </queuedValuePlacer>
</constructionHeuristic>
```

9.10. CHEAPEST INSERTION (CI)

9.10.1. アルゴリズムの説明

CI (Cheapest Insertion) アルゴリズムは、すべてのプランニングエンティティーに対して、すべての計画値を繰り返します。一度に 1 つのプランニングエンティティーを初期化します。(すべてのプランニングエンティティーおよび値から) 利用可能で最適な計画値を割り当て、すでに初期化されているプランニングエンティティーを考慮します。すべてのプランニングエンティティーが初期化されたら終了します。割り当ててからプランニングエンティティーが変更することはありません。



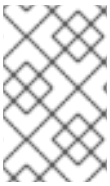
注記

CI (Cheapest Insertion) のスケールは、FF (First Fit) などに比べるとかなり悪いです。

9.10.2. 設定

CI (Cheapest Insertion) の最も簡単な設定方法:

```
<constructionHeuristic>
  <constructionHeuristicType>CHEAPEST_INSERTION</constructionHeuristicType>
</constructionHeuristic>
```



注記

InitializingScoreTrend を **ONLY_DOWN** にすると、このアルゴリズムは速くなります。エンティティーの場合は、そのスコアが直前のステップのスコアを悪化させない最初の Move を選択し、その後の Move は無視します。

高度な設定方法は、プールからの割り当て を参照してください。

9.11. REGRET INSERTION (RI)

9.11.1. アルゴリズムの説明

RI (Regret Insertion) アルゴリズムは、CI (Cheapest Insertion) アルゴリズムと同じように振舞います。すべてのプランニングエンティティーに、すべての計画値を繰り返します。一度に 1 つのプランニングエンティティーを初期化します。ただし、最高スコアを持つエンティティーと値の組み合わせを選択せず、最適解を割り当てるタイミングと次に最適な解を割り当てるタイミングの間でスコアロスが一番大きいエンティティーを選択します。次に、エンティティーを最適解に割り当てます。

9.11.2. 設定

このアルゴリズムは現在実装されていません。

9.12. ALLOCATE FROM POOL (AFP)

9.12.1. アルゴリズムの説明

AFP (Allocate From Pool) は、CI (Cheapest Insertion) および RI (Regret Insertion) に関する、用途が広い、一般的な形式です。以下のように振舞います。

1. プールに、エンティティーと値の組み合わせを追加する。
2. 最適エンティティーを最適値に割り当てる。
3. すべてのエンティティーが割り当てられるまで繰り返す。

9.12.2. 設定

簡単な設定方法:

```
<constructionHeuristic>
  <constructionHeuristicType>ALLOCATE_FROM_POOL</constructionHeuristicType>
</constructionHeuristic>
```

詳細で簡単な設定方法:

```
<constructionHeuristic>
  <constructionHeuristicType>ALLOCATE_FROM_POOL</constructionHeuristicType>
  <entitySorterManner>DECREASING_DIFFICULTY_IF_AVAILABLE</entitySorterManner>
  <valueSorterManner>INCREASING_STRENGTH_IF_AVAILABLE</valueSorterManner>
</constructionHeuristic>
```

entitySorterManner オプションと **valueSorterManner** オプションについては、 キューからエンティティーの割り当て を参照してください。

高度で詳細な設定方法: (変数を 1 つ持つエンティティークラス 1 つに CI (Cheapest Insertion) を設定):

```
<constructionHeuristic>
  <pooledEntityPlacer>
    <changeMoveSelector>
      <entitySelector id="placerEntitySelector">
        <cacheType>PHASE</cacheType>
        <selectionOrder>SORTED</selectionOrder>
        <sorterManner>DECREASING_DIFFICULTY</sorterManner>
      </entitySelector>
      <valueSelector>
```

```
        <cacheType>PHASE</cacheType>
        <selectionOrder>SORTED</selectionOrder>
        <sorterManner>INCREASING_STRENGTH</sorterManner>
      </valueSelector>
    </changeMoveSelector>
  </pooledEntityPlacer>
</constructionHeuristic>
```

ステップごとに、**PooledEntityPlacer** は、(**MoveSelector** が生成したそのエンティティーのすべての Move から) 勝利 **Move** を適用します。

エンティティーまたは値のソートをカスタマイズするには、ソートした選択 を参照してください。その他の セレクター のカスタマイズ (filtering や limiting など) も対応しています。

第10章 LOCAL SEARCH (局所探索法)

10.1. 概要

局所探索法は、初期解から始めて、1つの解をより良い解に進化させます。この探索法は、探索木ではなく、1つの解の探索パスを使用します。このパスにおける各解は、多くの解の Move を評価し、次の解への最も適した Move を適用し、終了するまで何度も繰り返します (通常は時間が切れるまで続きます)。

局所探索法の振舞いは、人間が計画する際の振る舞いにとても良く似ており、探索パスを1つ使用してファクトを移動し、実現の可能性が高い解を見つけます。したがって、これを実装するのがごく一般的です。

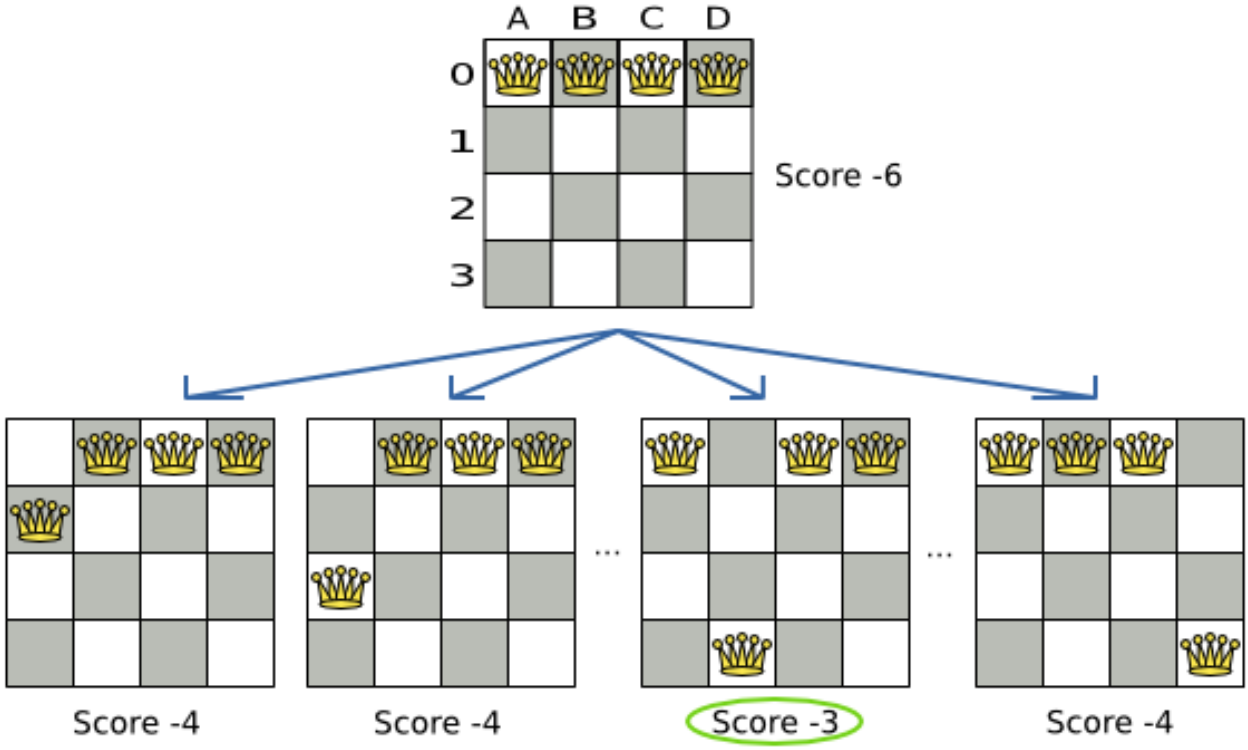
局所探索法は通常、初期解から開始する必要があるため、開始する前に構築ヒューリスティックの Solver フェーズを設定する必要があります。

10.2. 局所探索法の概念

10.2.1. 段階的

ステップは勝利 **Move** です。局所探索法は、現在の解に多くの Move を試し、最適な Move をステップとして選択します。

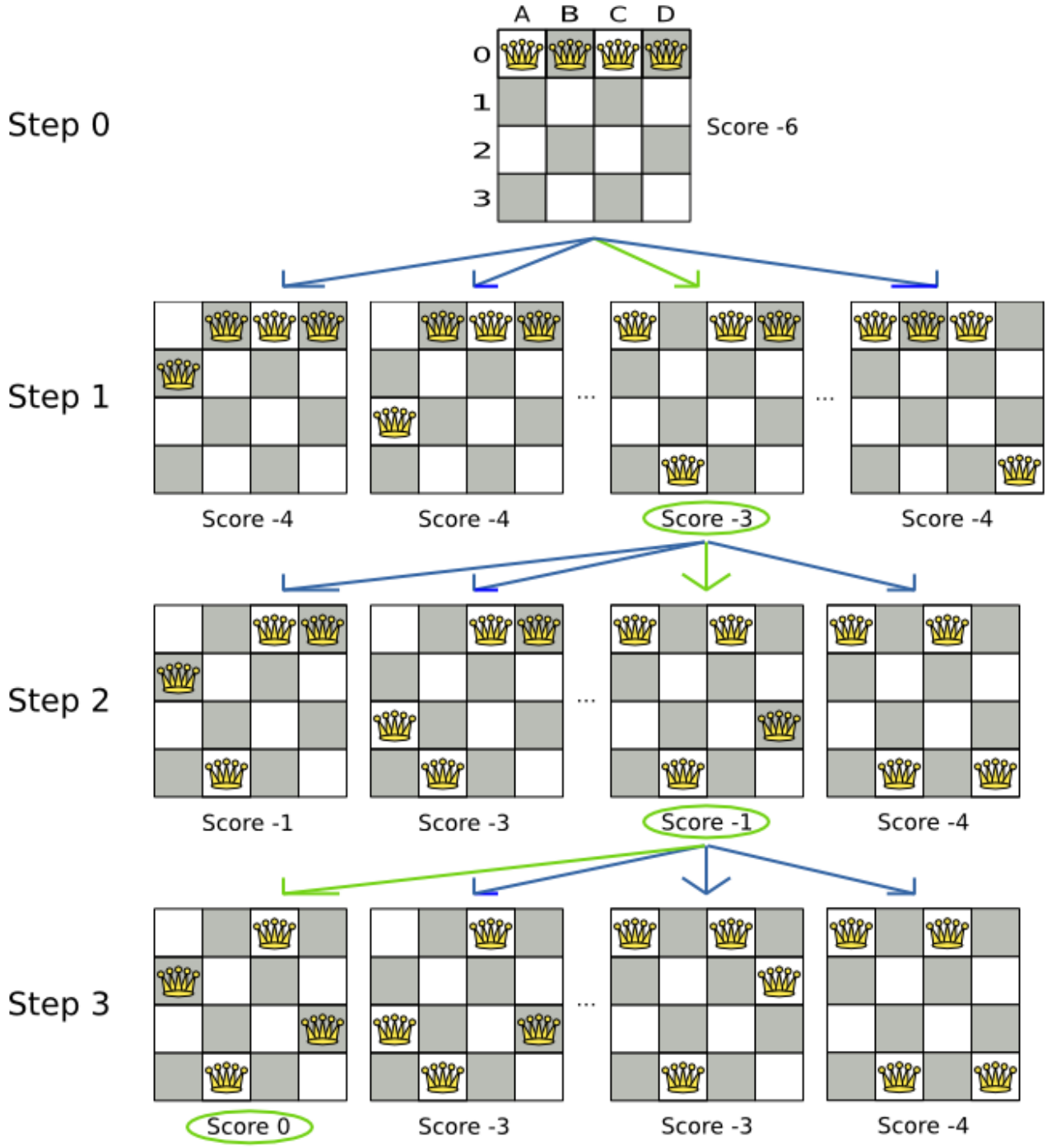
図10.1 ステップ 0 で次のステップの決定 (クイーン 4 個の例)



B0 から B3 への Move のスコアが一番高い (**-3**) ため、これが次のステップとして選択されます。最高スコアの Move が複数ある場合は、そのうちの1つがランダムに選択されます (この例では **B0** から **B3**)。(ここには示されていませんが) **C0** から **C3** への Move もスコアが **3** なので、代わりにこの Move が選択された可能性があります。

このステップが解に適用されたら、次の解に向けて、局所探索法がすべての Move をもう一度繰り返し、次のステップを決定します。これがループして継続していき、以下のような結果となります。

図10.2 全ステップ (クイーン 4 個の例)



局所探索法は、探索木ではなく、探索パスを使用している点に注意してください。探索パスは緑の矢印で示されます。各ステップで、選択された Move はすべて試されますが、次のステップとして選択されない限り、その解をそれ以上調べることはありません。これが、局所探索法が非常にスケーラブルな理由です。

上記の例が示すように、局所探索法は、クイーン 4 個の問題を、最初の解から始め、以下のステップを順次選択します。

1. B0 から B3
2. D0 から B2
3. A0 から B1

org.optaplanner カテゴリーの デバッグ ログを有効 にすると、ログにこのステップが示されます。

```
INFO  Solving started: time spent (0), best score (-6), random (JDK with seed 0).
DEBUG     LS step (0), time spent (20), score (-3), new best score (-3), accepted/selected move
count (12/12), picked move (Queen-1 {Row-0 -> Row-3}).
DEBUG     LS step (1), time spent (31), score (-1), new best score (-1), accepted/selected move
count (12/12), picked move (Queen-3 {Row-0 -> Row-2}).
DEBUG     LS step (2), time spent (40), score (0), new best score (0), accepted/selected move count
(12/12), picked move (Queen-0 {Row-0 -> Row-1}).
INFO  Local Search phase (0) ended: step total (3), time spent (41), best score (0).
INFO  Solving ended: time spent (41), best score (0), average calculate count per second (1780).
```

ログメッセージには、**Move** 実装の **toString()** メソッドが含まれます。これは、たとえば **Queen-1 {Row-0 -> Row-3}** を返します。

ネイティブの局所探索法の設定は、クイーン 4 個の問題を 3 つのステップで解きますが、可能解として評価するのは 37 個 (3 ステップで、1 ステップあたり 12 個の Move + 最初の解 1 つ) です。この値は、合計で 256 個となる可能解の一部でしかありません。クイーンが 16 個の場合は 31 ステップとなり、解の合計は 18446744073709551616 個になりますが、評価されるのは、そのうち 7441 個です。さらに、構築ヒューリスティック フェーズを最初に使用することで、効率がはるかに良くなります。

10.2.2. 次のステップを決定

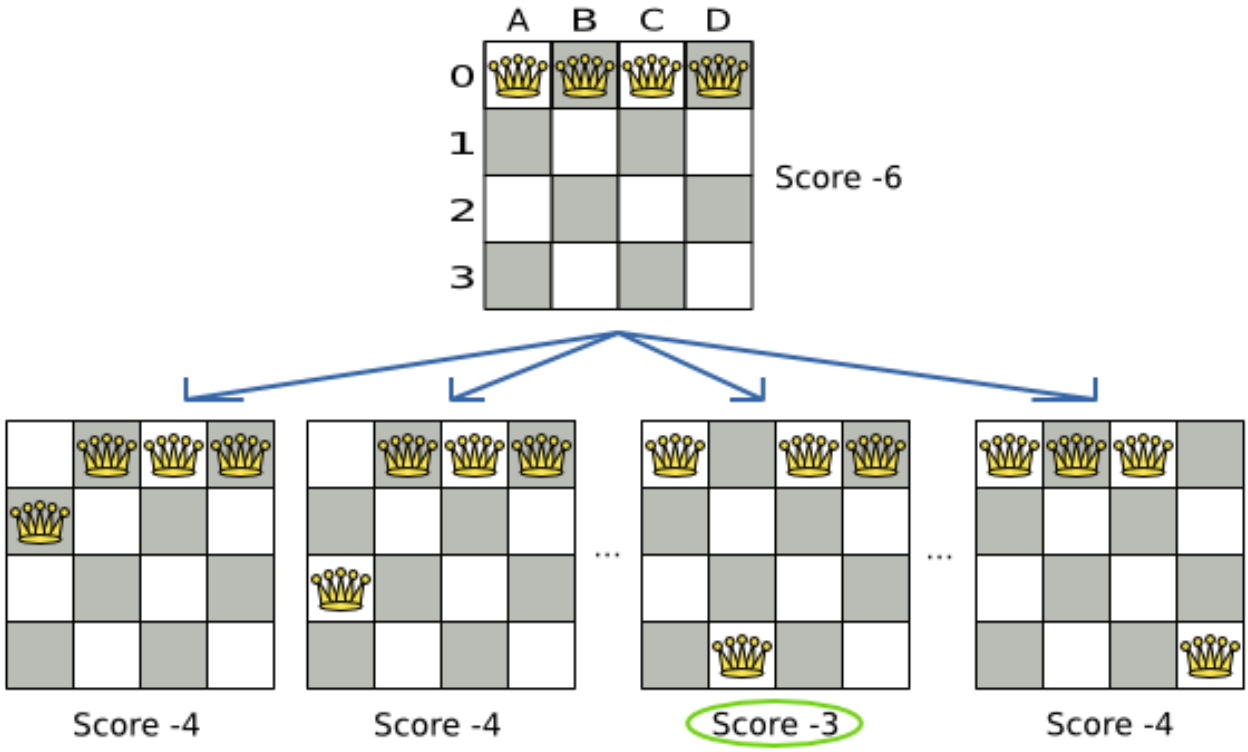
局所探索法は、次の設定可能なコンポーネントを 3 つ用いて、次のステップを決めます。

- 現在の解の可能な Move を選択する **MoveSelector**。Move と近傍選択 の章を参照してください。
- 認められない Move を除外する **Acceptor**。
- 認められた Move を集め、そこから次のステップを選択する **Forager**。

Solver フェーズ設定は以下のようになります。

```
<localSearch>
  <unionMoveSelector>
    ...
  </unionMoveSelector>
  <acceptor>
    ...
  </acceptor>
  <forager>
    ...
  </forager>
</localSearch>
```

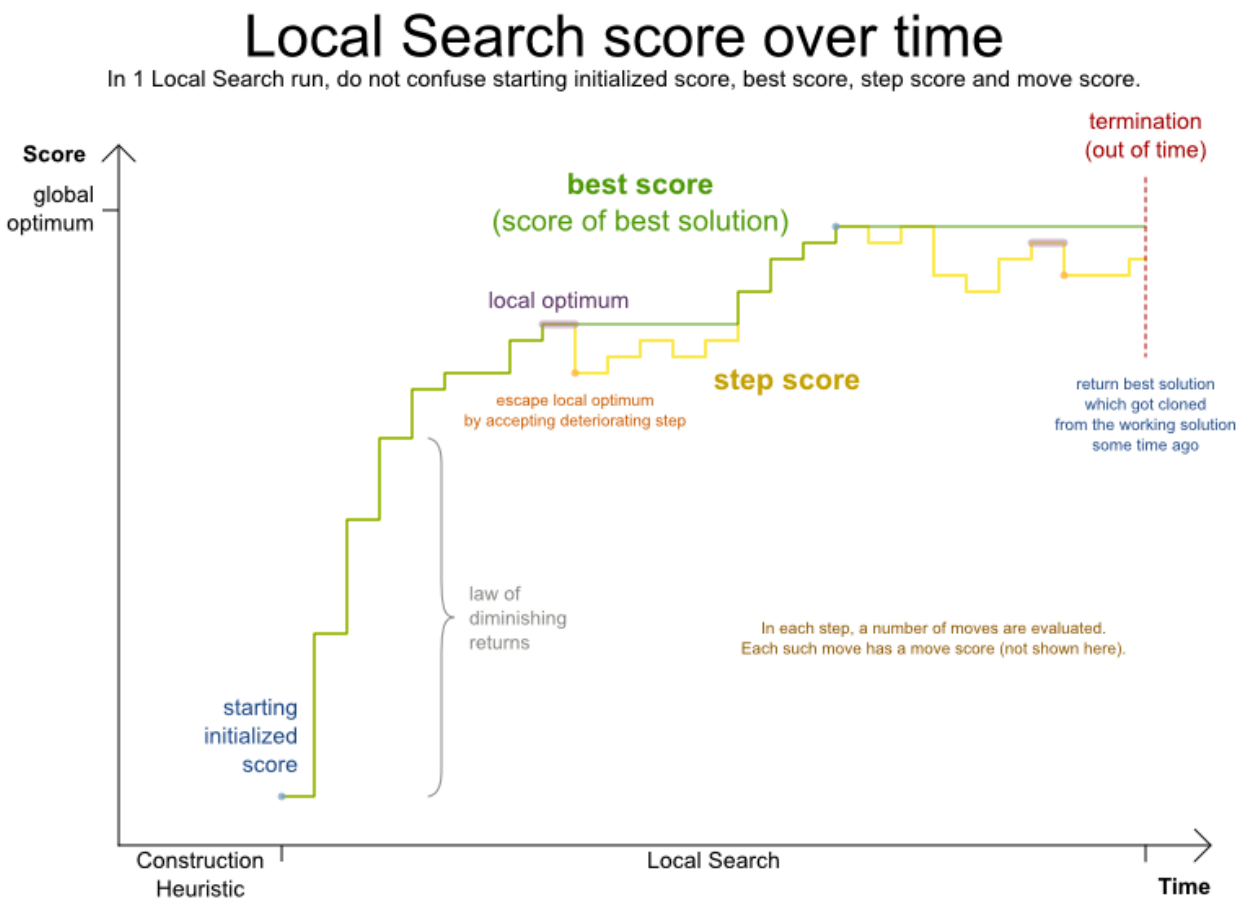
以下の例では、**MoveSelector** が、青い線で示される Move を集め、**Acceptor** がそのすべてを認め、**Forager** が B0 から B3 への Move を選択します。



トレース のログを有効にすると、ログに意思決定が示されます。

```
INFO  Solver started: time spent (0), score (-6), new best score (-6), random (JDK with seed 0).
TRACE        Move index (0) not doable, ignoring move (Queen-0 {Row-0 -> Row-0}).
TRACE        Move index (1), score (-4), accepted (true), move (Queen-0 {Row-0 -> Row-1}).
TRACE        Move index (2), score (-4), accepted (true), move (Queen-0 {Row-0 -> Row-2}).
TRACE        Move index (3), score (-4), accepted (true), move (Queen-0 {Row-0 -> Row-3}).
...
TRACE        Move index (6), score (-3), accepted (true), move (Queen-1 {Row-0 -> Row-3}).
...
TRACE        Move index (9), score (-3), accepted (true), move (Queen-2 {Row-0 -> Row-3}).
...
TRACE        Move index (12), score (-4), accepted (true), move (Queen-3 {Row-0 -> Row-3}).
DEBUG     LS step (0), time spent (6), score (-3), new best score (-3), accepted/selected move count
(12/12), picked move (Queen-1 {Row-0 -> Row-3}).
...
```

(たとえばタブー探索で) 最終解のスコアを下げることができるため、 **Solver** は、探索パス全体で見つけた最適解を記憶します。毎回、その時の解は、前の回の最適解を上回るため、その時点の解の クローンが作成され、新しい最適解として参照されます。



10.2.3. Acceptor

タブー探索、焼きなまし法、レイトアクセプタンスを有効にするために、(**Forager** とともに) **Acceptor** が使用されます。**Acceptor** は、各 Move に対し、それが認められるかどうかを確認します。

設定行をいくつか変更するだけで、タブー探索から、焼きなまし法またはレイトアクセプタンスに簡単に切り替え、元に戻すこともできます。

独自の **Acceptor** を実装することはできますが、組み込まれているもので大概のニーズに沿うはずです。複数の **Acceptor** を組み合わせることもできます。

10.2.4. Forager

Forager は、認められた Move をすべて集め、次のステップとなる Move を選択します。通常は、認められた Move の中で一番スコアが高いものが選択されます。スコアが一番高いものが複数ある場合は、その中からランダムに選択します。ランダムに選択することがより良い結果につながります。

注記

breakTieRandomly を **false** に設定すれば、ランダム選択することを無効にすることができますが、大抵の場合、無効にすることは推奨されません。

- 先の Move の方が、同じスコアを持つ後の Move より優れている場合は、スコア計算プログラムにより、よりソフトな スコアレベル が追加され、最初の Move のスコアをわずかに高くしています。これが強制的に行われる Move 選択順序には依存しないようにしてください。
- ランダムに選択することは、再現性 には影響しません。

10.2.4.1. 認められる数の制約

可能な Move が多数ある場合は、各ステップですべての Move を評価するのは非効率です。すべての Move でランダムなサブセットだけを評価するには、以下を使用します。

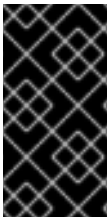
- 整数値 **acceptedCountLimit**。各ステップで認められた Move の中からいくつ評価するかを指定します。デフォルトでは、各ステップで、認められたすべての Move が評価されます。

```
<forager>
  <acceptedCountLimit>1000</acceptedCountLimit>
</forager>
```

N クィーンの問題とは異なり、現実世界の問題では **acceptedCountLimit** を使用する必要があります。**acceptedCountLimit** を 2 秒以内に 1 ステップ行うことからはじめます。 デバッグ ログを有効 にし、ステップの回数を確認します。

```
DEBUG     LS step (0), time spent (20), score (-3), new best score (-3), ...
```

ベンチマーカー を使用して、値を調整します。



重要

acceptedCountLimit を低く (つまり、ステップ移動のアルゴリズムを速く) した場合、**selectionOrder SHUFFLED** は使用しないようにしてください。シャッフルを行うことで、セレクターの全要素にランダムな値を生成するため時間がかかりますが、実際に選択される要素はほんのわずかになるからです。

10.2.4.2. 早期発見のタイプ

forager はステップの早い段階で Move を選択し、その後に選択される Move を無視することができます。局所探索法には、3 つの早期発見のタイプがあります。

- **NEVER**: Move が早期に選択されることはありません。その選択が許可する Move がすべて評価されます。これがデフォルトです。

```
<forager>
  <pickEarlyType>NEVER</pickEarlyType>
</forager>
```

- **FIRST_BEST_SCORE_IMPROVING**: 最高スコアを上回る、最初に許可された Move を選択します。最高スコアを上回るものがなければ、**pickEarlyType NEVER** と同じ振舞いになります。

```
<forager>
  <pickEarlyType>FIRST_BEST_SCORE_IMPROVING</pickEarlyType>
</forager>
```

- **FIRST_LAST_STEP_SCORE_IMPROVING**: 最後のステップスコアを上回る、最初に許可された Move を選択します。最後のステップスコアを上回るものがなければ、**pickEarlyType NEVER** と同じ振舞いになります。

```
<forager>
  <pickEarlyType>FIRST_LAST_STEP_SCORE_IMPROVING</pickEarlyType>
</forager>
```

10.3. HILL CLIMBING (山登り法) (SIMPLE LOCAL SEARCH: 簡易局所探索法)

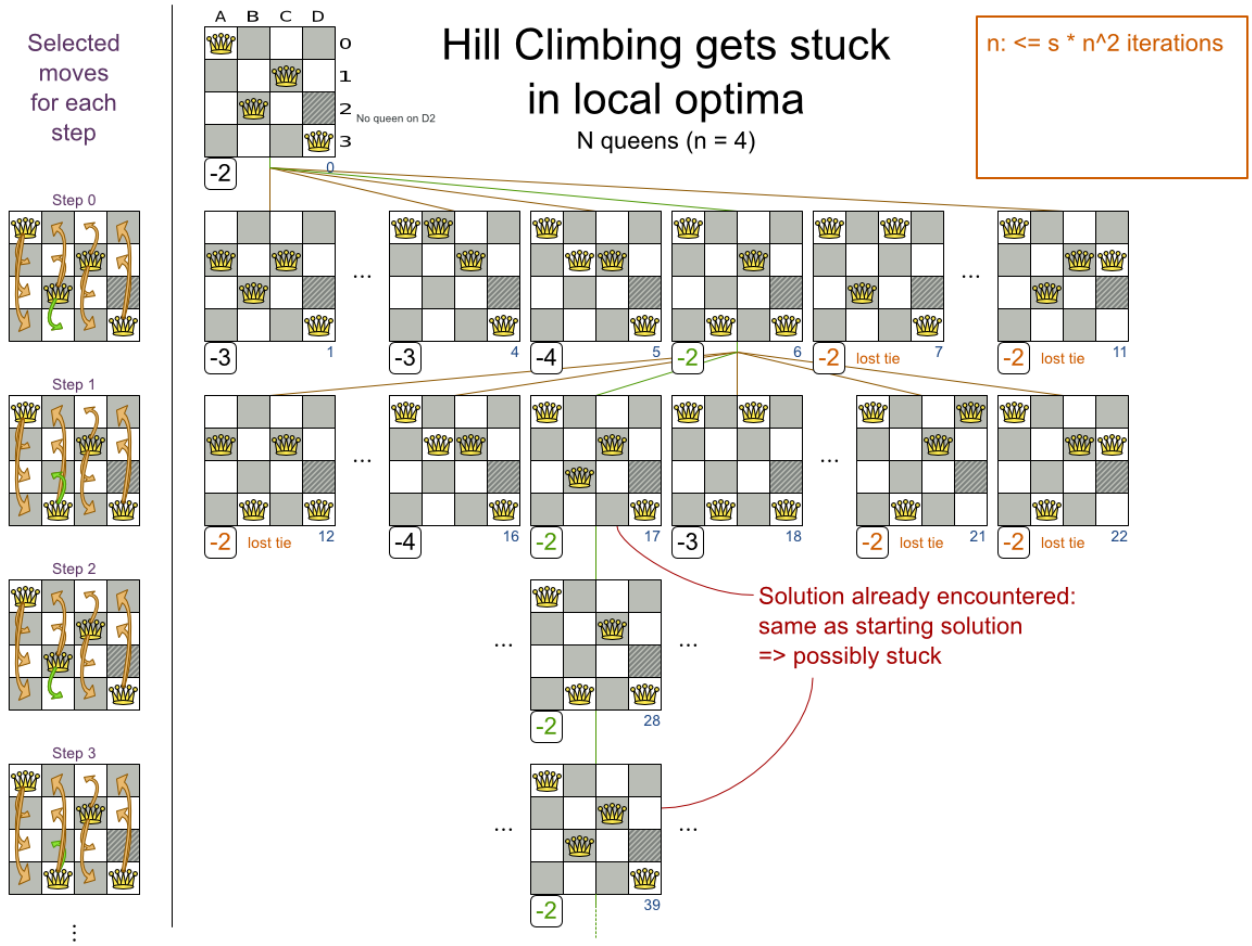
10.3.1. アルゴリズムの説明

山登り法は、選択したすべての Move を試し、スコアが一番高い解を生む最適な Move を選択します。最適な Move は、ステップ Move と呼ばれます。その新しい解から、再度選択したすべての Move を試し、最適な Move を選択し、それを繰り返します。最適な Move が複数ある場合は、そのうちの 1 つがランダムに選択されます。

クイーンが動いたら、再度動かせます。NP 完全問題では、プランニング変数に最適な最終値を予測することは不可能なので、これは利点となります。

10.3.2. 局所最適条件での行き詰まり

山登り法は改善する Move を常に取得します。これは利点であるように思われますが、実際は利点ではありません。山登り法は局所最適条件で簡単に行き詰まる可能性があります。これは、すべての Move がスコアを下げる解にたどり着くと発生します。いずれの Move を選択しても、次のステップで元の解に戻り、無限ループとなります。



山登り法における改善 (タブー探索、焼きなまし法、レイトアクセプタンスなど) は、局所最適条件で行き詰まる問題に対応します。したがって、計画問題に局所最適条件がないことが明らかな場合を除いて、山登り法を使用することは推奨されません。

10.3.3. 設定

最も簡単な設定方法:

```
<localSearch>
  <localSearchType>HILL_CLIMBING</localSearchType>
</localSearch>
```

高度な設定方法:

```
<localSearch>
  ...
  <acceptor>
    <acceptorType>HILL_CLIMBING</acceptorType>
  </acceptor>
  <forager>
```

```
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

10.4. TABU SEARCH (タブー探索)

10.4.1. アルゴリズムの説明

タブー探索は、山登り法と同じように機能しますが、局所最適条件で行き詰まらないようにタブーリストを維持します。タブーリストには、直近で使用され、今は使用が禁止されるオブジェクトが保持されています。タブーリストのオブジェクトに関連する Move は容認されません。タブーリストのオブジェクトは、Move に関するもの (プランニングエンティティー、計画値、Move、解など) になります。以下は、クイーン 4 個でエンティティータブーを持つ例で、このクイーンはタブーリストに入っています。

注記

タブー検索の英語のスペルは、Taboo Search ではなく Tabu Search です。

学術論文: **Tabu Search - Part 1 and Part 2**by Fred Glover (1989 - 1990)

10.4.2. 設定

最も簡単な設定方法:

```
<localSearch>
  <localSearchType>TABU_SEARCH</localSearchType>
</localSearch>
```

タブー検索がステップを選択していくと、1つまたは複数のタブーが作成されます。多くのステップでは、その Move がタブーに違反する場合は Move が許可されません。ステップ数がタブーのサイズになります。高度は設定方法は以下のようになります。

```
<localSearch>
  ...
  <acceptor>
    <entityTabuSize>7</entityTabuSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1000</acceptedCountLimit>
  </forager>
</localSearch>
```

重要

タブー探索アクセプターでは、**acceptedCountLimit** を高く (**1000** など) 設定する必要があります。

Planner には、タブーのタイプが複数あります。

- プランニングエンティティーのタブー (推奨) は、直近のステップタブーのプランニングエンティティーを作成します。たとえば、N クィーンの場合は、直近に移動したクィーンのタブーが作成されます。まずはこのタブータイプを使用することが推奨されます。

  ```
  <acceptor>
    <entityTabuSize>7</entityTabuSize>
  </acceptor>
  ```

  タブーサイズをハードコードしないようにするには、エンティティー数に対するタブーの割合 (2% など) を設定します。

  ```
  <acceptor>
    <entityTabuRatio>0.02</entityTabuRatio>
  </acceptor>
  ```

- 計画値のタブー は、直近のステップのタブーのプランニングエンティティーを作成します。たとえば、N クィーンの場合は、直近に行を移動したクィーンをタブーにします。

  ```
  <acceptor>
    <valueTabuSize>7</valueTabuSize>
  </acceptor>
  ```

  タブーサイズをハードコードしないようにするには、値に対するタブーの割合 (2% など) を設定します。

  ```
  <acceptor>
    <valueTabuRatio>0.02</valueTabuRatio>
  </acceptor>
  ```

- **Move のタブー** は、直近のステップのタブーを選択します。そのステップの 1 に等しい Move は許可されません。

```
<acceptor>
  <moveTabuSize>7</moveTabuSize>
</acceptor>
```

- **Move の取り消しタブー** は、直近に移動した Move の取り消しがタブーになります。

```
<acceptor>
  <undoMoveTabuSize>7</undoMoveTabuSize>
</acceptor>
```

- 解のタブーは、直近にたどり着いた解をタブーにします。そのいずれかの解に向かう Move は許可されません。解 (ソリューション) に **equals()** と **hashCode()** が適切に実装されている必要があります。メモリーに余裕がある場合は、このタブーのサイズを大きくします。

```
<acceptor>
  <solutionTabuSize>1000</solutionTabuSize>
</acceptor>
```

  重要なケースでは、解のタブーは通常役に立ちません。探索領域サイズ により、同じ解に 2 回到達するには統計的にまずありえません。したがって、データセットが小さい場合を除き、これを使用することは推奨されません。

タブーの種類を複数組み合わせると便利な場合があります。

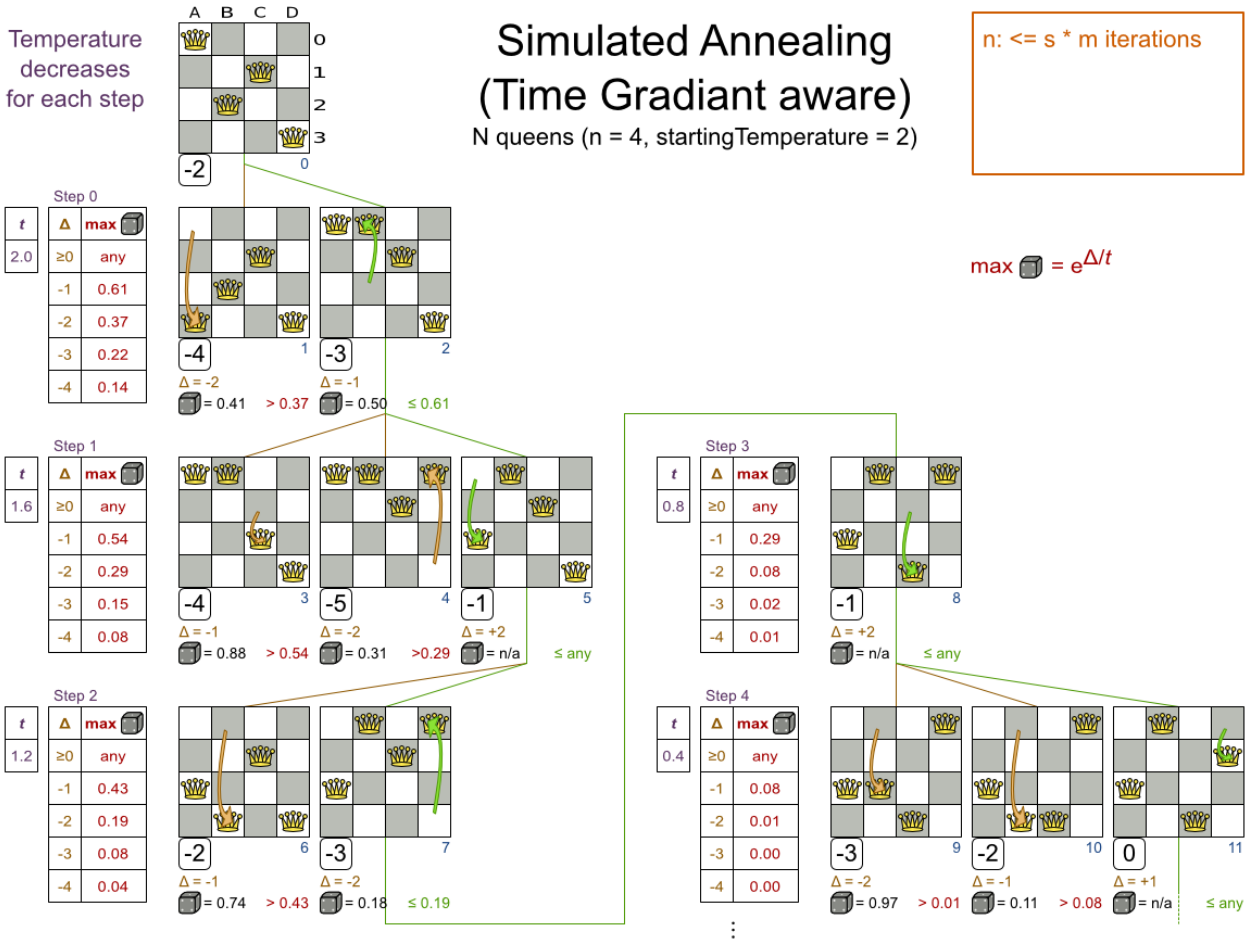
```
<acceptor>
  <entityTabuSize>7</entityTabuSize>
  <valueTabuSize>3</valueTabuSize>
</acceptor>
```

タブーのサイズが小さすぎると、Solver は局所最適条件で行き詰まる可能性があります。逆に、タブーのサイズが大きすぎると、壁で跳ね返って役に立たない可能性があります。ベンチマーカー を使用して、設定を適切に調整します。

10.5. SIMULATED ANNEALING (焼きなまし法)

10.5.1. アルゴリズムの説明

焼きなまし法は、ステップごとに数 Move だけを評価するため、すぐに次に進みます。標準的な実装では、最初に許可された Move が勝利ステップになります。Move は、スコアを減らさない場合は許可されます。スコアを減らした場合は、ランダムチェックを通過します。スコアを減らす Move がランダムチェックを通る可能性は、スコアのデクリメントの大きさと、フェーズが実行している時期 (度合い (temperature) として表現されます) に応じて減ります。



焼きなまし法は、スコアが一番高い Move を常に選択するわけではありません。また、1 ステップに対して Move を多数評価するわけでもありません。少なくても最初はそのように振舞います。改善しない Move が選択される可能性もありますが、これは、その Move のスコアと、終了 の時間勾配によります。最終的には、次第に山登り法になっていき、改善する Move だけを許可します。

10.5.2. 設定

まずは、**simulatedAnnealingStartingTemperature** を、1 つの Move が原因となる最大スコアデルタに設定します。ベンチマーカー で値を調整します。以下は、高度な設定になります。

```
<localSearch>
  ...
  <acceptor>

<simulatedAnnealingStartingTemperature>2hard/100soft</simulatedAnnealingStartingTemperature>
  </acceptor>
  <forager>
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

焼きなまし法では、**acceptedCountLimit** は低くする必要があります。標準的なアルゴリズムでは、**acceptedCountLimit** を **1** にしますが、**4** にした方がパフォーマンスが高いこともしばしばあります。

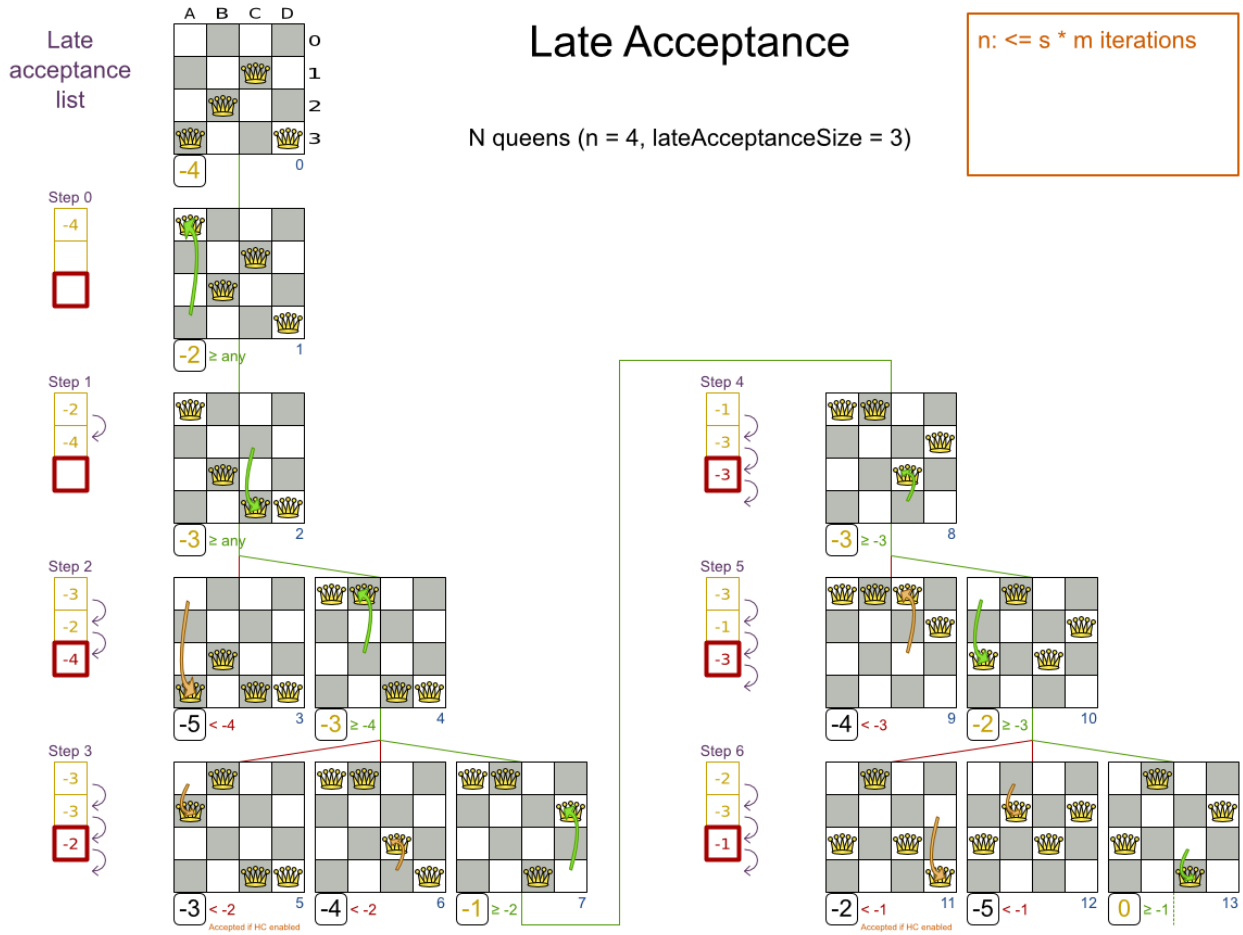
焼きなまし法は、タブーアクセプターと一緒に使用することができます。これにより、焼きなまし法が少し良くなります。タブー探索だけを使用する場合と比べて、タブーのサイズを小さくします。

```
<localSearch>
  ...
  <acceptor>

<simulatedAnnealingStartingTemperature>2hard/100soft</simulatedAnnealingStartingTemperature>
    <entityTabuSize>5</entityTabuSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

10.6. LATE ACCEPTANCE (レイトアクセプタンス)

10.6.1. アルゴリズムの説明

レイトアクセプタンス (LAHC (Late Acceptance Hill Climbing) とも呼ばれています) は、ステップごとに数 Move だけを評価します。スコアが減らない場合、もしくは最後のスコアと同じスコアになった (一定の前ステップでは勝利スコアだったもの) 場合は、Move が許可されます。



学術論文: The Late Acceptance Hill-Climbing Heuristic by Edmund K. Burke, Yuri Bykov (2012)

10.6.2. 設定

最も簡単な設定方法:

-

```
<localSearch>
  <localSearchType>LATE_ACCEPTANCE</localSearchType>
</localSearch>
```

レイトアクセプタンスは、何ステップも前に最適スコアだったものよりも高いスコアを持つ Move を許可します。このステップの数は **lateAcceptanceSize** です。以下は高度な設定になります。

```
<localSearch>
  ...
  <acceptor>
    <lateAcceptanceSize>400</lateAcceptanceSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

レイトアクセプタンスでは、**acceptedCountLimit** を低くする必要があります。

レイトアクセプタンスは、タブアクセプターと一緒に使用することができます。これにより、レイトアクセプタンスが少し良くなります。タブー探索を使用する場合と比べて、タブーのサイズを小さくします。

```
<localSearch>
  ...
  <acceptor>
    <lateAcceptanceSize>400</lateAcceptanceSize>
    <entityTabuSize>5</entityTabuSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

10.7. STEP COUNTING HILL CLIMBING (SCHC)

10.7.1. アルゴリズムの説明

SCHC (Step Counting Hill Climbing) も、ステップごとに数 Move だけ評価します。ステップのスコアをしきい値として保持します。スコアが減らない場合、またはスコアがそのしきい値以上になる場合は、Move が許可されます。

学術論文: An initial study of a novel Step Counting Hill Climbing heuristic applied to timetabling problems by Yuri Bykov, Sanja Petrovic (2013)

10.7.2. 設定

SCHC (Step Counting Hill Climbing) は、しきい値スコアよりも高いスコアを持つ Move だけを許可します。(**stepCountingHillClimbingSize** で指定した) すべてのステップで、しきい値はステップのスコアに設定されます。

```
<localSearch>
  ...
```

```
  <acceptor>
    <stepCountingHillClimbingSize>400</stepCountingHillClimbingSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1</acceptedCountLimit>
  </forager>
</localSearch>
```

SCHC (Step Counting Hill Climbing) では、**acceptedCountLimit** を低くする必要があります。

SCHC (Step Counting Hill Climbing) は、レイトアクセプタンスセクション で説明しているように、タブーアクセプターと同時に使用することができます。

10.8. STRATEGIC OSCILLATION (SO)

10.8.1. アルゴリズムの説明

SO (Strategic Oscillation) は、特に タブー探索 と使用すると効果があるアドオンです。スコアが一番高い、許可された Move を選択するのではなく、別のメカニズムを使用します。改善される Move がある場合は、それを選択しますが、ない場合は、よりハードなスコアを壊すのが少ない Move よりは、よりソフトなスコアレベルを改善する Move を選択します。

10.8.2. 設定

以下の例では、タブー探索設定で **finalistPodiumType** を設定します。

```
<localSearch>
  ...
  <acceptor>
    <entityTabuSize>7</entityTabuSize>
  </acceptor>
  <forager>
    <acceptedCountLimit>1000</acceptedCountLimit>
    <finalistPodiumType>STRATEGIC_OSCILLATION</finalistPodiumType>
  </forager>
</localSearch>
```

以下の **finalistPodiumTypes** がサポートされます。

- **HIGHEST_SCORE** (デフォルト): 許可された Move の中でスコアが一番高いものを選択します。
- **STRATEGIC_OSCILLATION**: デフォルトの SO (Strategic Oscillation) バリアントに対するエイリアスです。
- **STRATEGIC_OSCILLATION_BY_LEVEL**: 許可されている、改善する Move がある場合はそれを選択します。そのような Move がない場合は、許可された Move の中から、よりソフトなスコアレベルを改善するものを選択します (ハードなスコアレベルは考慮されません)。最後に完了したステップスコアよりも良くなると、Move は改善しています。
- **STRATEGIC_OSCILLATION_BY_LEVEL_ON_BEST_SCORE**: **STRATEGIC_OSCILLATION_BY_LEVEL** と同じですが、(最後に完了したステップのスコアではなく) 最高スコアよりも良くなるように定義します。

10.9. カスタムの終了、MOVESELECTOR、ENTITYSELECTOR、VALUESELECTOR、またはアクセプターの使用

抽象型クラスと、それに関連する ***Config** クラスを拡張して、カスタムの 終了、**MoveSelector**、**EntitySelector**、**ValueSelector**、または アクセプター を組み込むことができます。

たとえば、カスタムの **MoveSelector** を使用するには、**AbstractMoveSelector** クラスを拡張し、**MoveSelectorConfig** クラスを拡張してSolver 設定で設定します。

注記

以下の理由により、(**Config** クラスも拡張するのを避けるために) 終了 などのインスタンスを直接追加することはできません。

- **SolverFactory** は、複数の **Solver** インスタンスを構築しますが、これにはそれぞれ、**Termination** などのインスタンスが必要です。
- Solver 設定では、XML とのやり取りをシリアル化する必要があります。これにより、XML で複数の **Solver** バリアントを設定できるため、**PlannerBenchmark** を使用したベンチマーク作成が特に簡単になります。
- **Config** クラスは、設定がより簡単で明確になることが多々あります。たとえば、**TerminationConfig** は、**minutesSpentLimit** と **secondsSpentLimit** を **timeMillisSpentLimit** に変換します。

ドメイン固有ではなく、より良い実装を構築する場合は、それを GitHub での pulll リクエストとして還元することを検討してください。それを最適化し、今後のリファクタリングに採用します。

第11章 EVOLUTIONARY ALGORITHMS (進化アルゴリズム)

11.1. 概要

進化アルゴリズムは、解の集団に有効で、その集団を進化させます。

11.2. EVOLUTIONARY STRATEGIES (進化ストラテジー)

このアルゴリズムは現在実装されていません。

11.3. GENETIC ALGORITHMS (遺伝的アルゴリズム)

このアルゴリズムは現在実装されていません。

注記

Planner では、しばらく前に優れた遺伝的アルゴリズムが記述されましたが、その時点ではマージしてサポートするのは現実的ではありませんでした。遺伝的アルゴリズムの結果は、すべてのユースケースを試した他の種類のすべての局所探索法 (山登り法を除く) と比べて常に著しく劣ります。にもかかわらず、Planner の将来バージョンでは、遺伝的アルゴリズムへの対応が追加されるため、お使いのユースケースで遺伝的アルゴリズムを、基準に従って簡単に評価することができます。

第12章 HYPERHEURISTICS (ハイパーヒューリスティック)

12.1. 概要

ハイパーヒューリスティックは、特定のデータセットで使用するヒューリスティックを自動的に決定します。

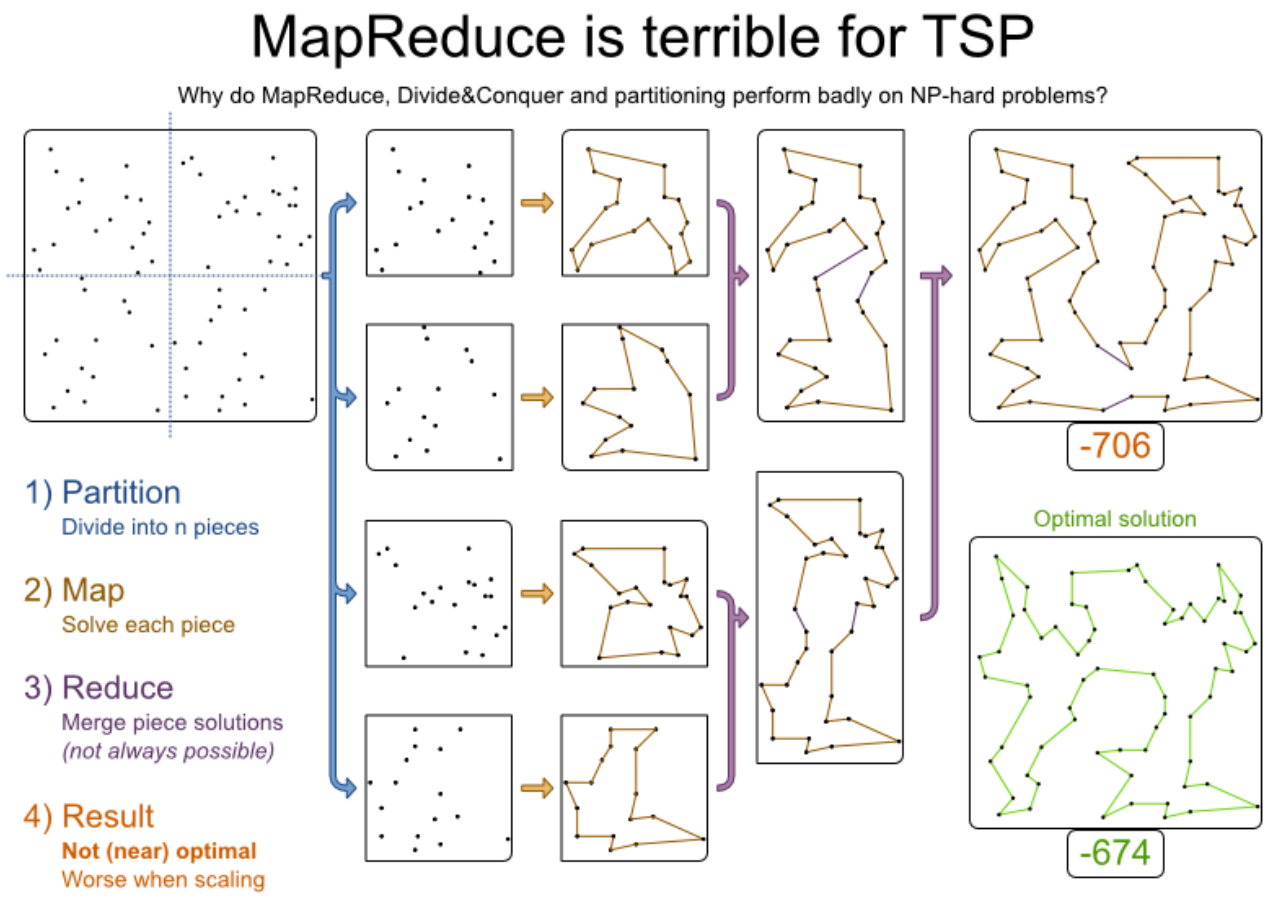
Planner の将来バージョンでは、ハイパーヒューリスティックにネイティブサポートを提供しますが、手動でも簡単に実装できます。データセットのサイズまたは難易度 (基準になっています) によって、使用する Solver 設定が異なります (もしくは、Solver 設定 API を使用してデフォルト設定を調整します)。この基準については、 ベンチマーカー で確認できます。

第13章 PARTITIONED SEARCH (分割検索)

13.1. 概要

データセットが非常に大きい場合は、データセットを小さく分割すると役に立つ場合があります。

ただし、分割を行うと、分割した各データセットでは最適な解決が得られても、全体としては十分ではなくなる場合があります。



Planner の今後のバージョンでは、数種類の分割をネイティブサポートしますが、上の図のように、手動でも実装することができます。分割したデータの 1 つを解決するには、**Solver** を使用してください。

注記

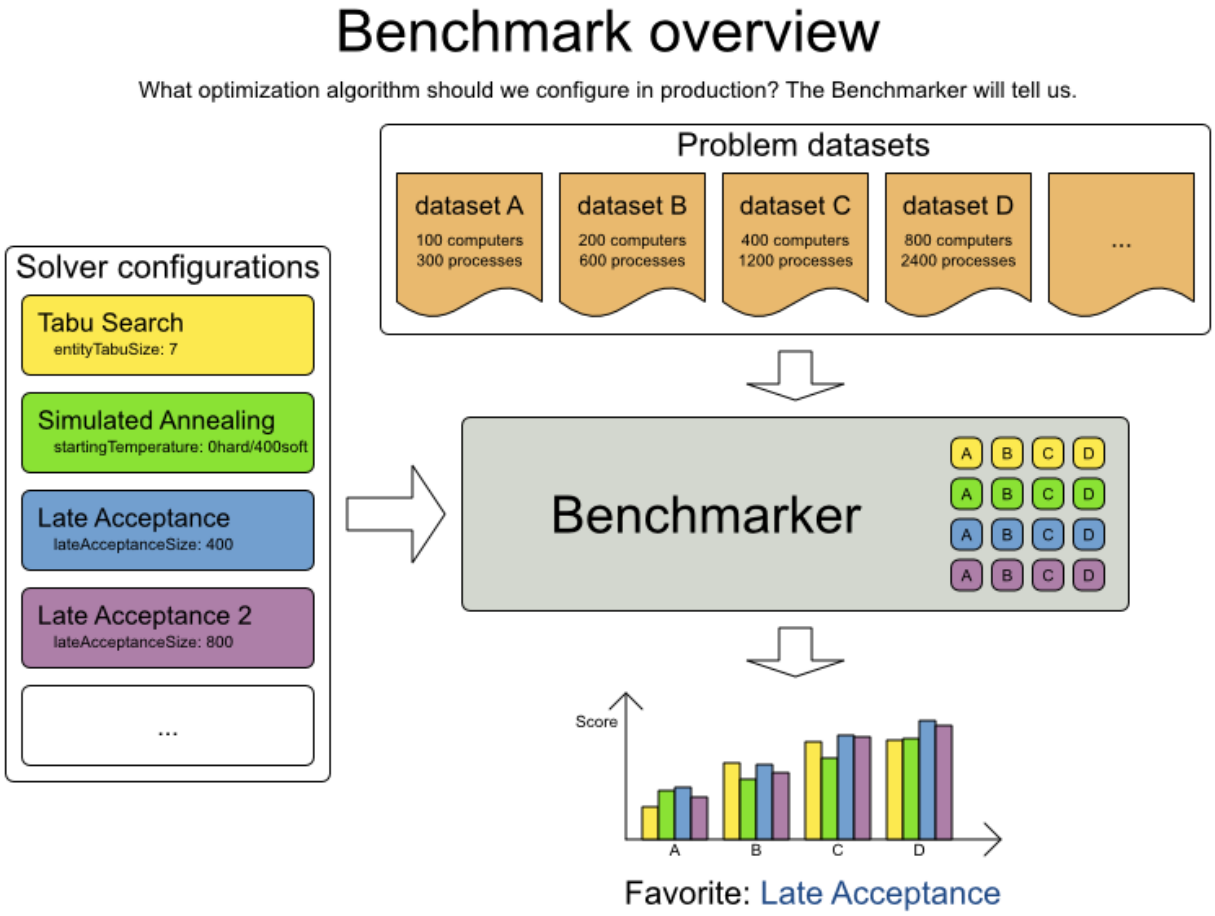
すべてのユースケースを分割できるわけではありません。これは、プランニングエンティティーと、値の範囲を n 個に分割できるユースケースでのみ有効です。たとえば、制約は、分割した各データの境界を超えることはできません。

第14章 ベンチマークおよび調整

14.1. 最適な SOLVER 設定を見つける

Planner は、複数の最適化アルゴリズムに対応しているため、おそらく、どのアルゴリズムが最適なのか疑問に思うはずです。一般的に、他のと比べて良い結果を得られる最適化アルゴリズムを選択することはできますが、良い結果を得られるかどうかは問題のドメインに大きく依存します。多くの Solver フェーズには、調整可能なパラメーターが含まれます。ほとんどの場合、設定せずに使用してもかなり良い結果は得られますが、パラメーターを使用することで結果が大きく変わります。

このため、Planner にはベンチマーカーが用意されています。これを使用して、開発過程で、異なる Solver フェーズに異なる設定を試すことができるため、本番環境での計画問題に最適な設定を使用できます。



14.2. ベンチマーク設定

14.2.1. optaplanner-benchmark に依存関係を追加

このベンチマーカーは、**optaplanner-benchmark** という名前の別のアーティファクトにあります。

Maven を使用する場合は、**pom.xml** ファイルに依存関係を追加します。

```
<dependency>
  <groupId>org.optaplanner</groupId>
  <artifactId>optaplanner-benchmark</artifactId>
```

```
    </dependency>
```

これは、Gradle、Ivy、および Buildr と同じです。このバージョンは、使用される **optaplanner-core** バージョンと完全に同じです (**optaplanner-bom** をインポートする場合は自動的にこのようになります)。

Ant を使用している場合は、おそらく、ダウンロードした Zip の **binaries** ディレクトリーから、必要な JAR をコピーしています。

14.2.2. PlannerBenchmark の構築と実行

PlannerBenchmarkFactory で、**PlannerBenchmark** インスタンスを構築します。クラスパスリソースとして提供されている、XML のベンチマーク設定ファイルを使用して設定します。

```
        PlannerBenchmarkFactory plannerBenchmarkFactory =
PlannerBenchmarkFactory.createFromXmlResource(
                "org/optaplanner/examples/nqueens/benchmark/nqueensBenchmarkConfig.xml");
        PlannerBenchmark plannerBenchmark = plannerBenchmarkFactory.buildPlannerBenchmark();
        plannerBenchmark.benchmark();
```

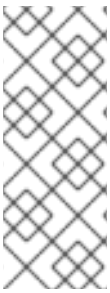
ベンチマーク設定は、以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens</benchmarkDirectory>

  <inheritedSolverBenchmark>
    <problemBenchmarks>
      ...
      <inputSolutionFile>data/nqueens/unsolved/32queens.xml</inputSolutionFile>
      <inputSolutionFile>data/nqueens/unsolved/64queens.xml</inputSolutionFile>
    </problemBenchmarks>
    <solver>
      ...<!-- Common solver configuration -->
    </solver>
  </inheritedSolverBenchmark>

  <solverBenchmark>
    <name>Tabu Search</name>
    <solver>
      ...<!-- Tabu Search specific solver configuration -->
    </solver>
  </solverBenchmark>
  <solverBenchmark>
    <name>Simulated Annealing</name>
    <solver>
      ...<!-- Simulated Annealing specific solver configuration -->
    </solver>
  </solverBenchmark>
  <solverBenchmark>
    <name>Late Acceptance</name>
    <solver>
      ...<!-- Late Acceptance specific solver configuration -->
```

```
    </solver>
  </solverBenchmark>
</plannerBenchmark>
```

この **PlannerBenchmark** は、2 つのデータセット (32queens および 64queens) を使用して、3 つの設定 (タブー探索、焼きなまし法、およびレイトアクセプタンス) を試すため、Solver を全部で 6 回実行します。

すべての **<solverBenchmark>** 要素には、Solver 設定と、1 つ以上の **<inputSolutionFile>** 要素があります。未解決の解の各ファイルで、Solver 設定を実行します。**name** 要素はオプションです (ない場合は生成されます)。**inputSolutionFile** は、SolutionFileIO (ワーキングディレクトリーへの相対パス) から読み込まれます。

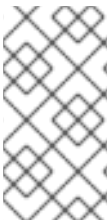


注記

(たとえば **<inputSolutionFile>** 要素で) フォワードスラッシュ (/) をファイルセパレーターとして使用します。これは、(Windows を含む) どのプラットフォームでも有効です。

ファイルセパレーターにバックスラッシュ (\) は使用しないでください。Linux と Mac では機能しないため、移植性がなくなります。

ベンチマークレポートは、**<benchmarkDirectory>** 要素 (ワーキングディレクトリーへの相対パス) で指定したディレクトリーに記述されます。



注記

benchmarkDirectory は、ソース制御の範囲外で、ビルドシステムで削除されないディレクトリーにすることが推奨されます。この方法を使用すれば、生成したファイルによってソース制御が大きくなることはなく、クリーンビルドを行っても失われることがありません。通常、このディレクトリーは ローカル と呼ばれます。

Exception または **Error** が 1 つのベンチマークで発生したとき、(Planner のその他の機能とは異なり) ベンチマーカー全体でフェイルファーストにはなりません。代わりに、ベンチマーカーは、その他のすべてのベンチマークを実行し続け、ベンチマークレポートを作成してからフェイルします (ベンチマークが 1 つでも失敗した場合)。失敗したベンチマークは、ベンチマークレポートに記録されます。

14.2.2.1. 継承される Solver ベンチマーク

詳細のレベルを下げるために、複数の **<solverBenchmark>** 要素で共通している部分が **<inheritedSolverBenchmark>** 要素に抽出されます。すべてのプロパティーは **<solverBenchmark>** 要素で上書きすることができますが、**<constructionHeuristic>** や **<localSearch>** などの継承された Solver フェーズは上書きされず、Solver フェーズリストの最後に追加されます。

14.2.3. SolutionFileIO: ソリューションファイルの入出力

14.2.3.1. SolutionFileIO インターフェース

ベンチマーカーは、ソリューション をロードするために、入力ファイルを読み込めるようにする必要があります。また、各ベンチマークで最適な ソリューション を出力ファイルに書き込むことも必要になるかもしれません。その場合は、**SolutionFileIO** インターフェースを実装するクラスを使用します。

```
public interface SolutionFileIO {
```

```
    String getInputFileExtension();

    String getOutputFileExtension();

    Solution read(File inputSolutionFile);

    void write(Solution solution, File outputSolutionFile);

}
```

SolutionFileIO インターフェースは **optaplanner-persistence-common** JAR (**optaplanner-benchmark** JAR の依存関係) にあります。

14.2.3.2. XStreamSolutionFileIO: デフォルトの SolutionFileIO

デフォルトでは、ベンチマーカーは **XStreamSolutionFileIO** インスタンスを使用して、ソリューションの読み書きを行います。

XStream アノテーションでアノテートした **Solution** クラスについて、ベンチマーカーに指定する必要があります。

```
<problemBenchmarks>

<xStreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</xStreamAnnotatedClass>
  <inputSolutionFile>data/nqueens/unsolved/32queens.xml</inputSolutionFile>
  ...
</problemBenchmarks>
```

XStreamSolutionFileIO は、カスタマイズした **XStream** インスタンスを使用するため、このような入力ファイルでは、**XStream** インスタンスではなく、**XStreamSolutionFileIO** インスタンスで指定します。

警告

XStream (および一般的には XML) は、非常に詳細な書式を使用しています。このフォーマットに非常に長いデータセットを読み込んだり書き込んだりすると、**OutOfMemoryError** が発生したり、パフォーマンスが低下する原因になります。

14.2.3.3. カスタムの SolutionFileIO

その他に、カスタマイズした **SolutionFileIO** 実装を作成し、**solutionFileIOClass** 要素で設定します。

```
<problemBenchmarks>

<solutionFileIOClass>org.optaplanner.examples.machinereassignment.persistence.MachineReassignmentFileIO</solutionFileIOClass>
```

```
    <inputSolutionFile>data/machinereassignment/import/model_a1_1.txt</inputSolutionFile>
    ...
  </problemBenchmarks>
```

出力ファイルを入力ファイルとして読み込むことが推奨されます。これは、**getInputFileExtension()** と **getOutputFileExtension()** が同じ値を返すことを意味します。

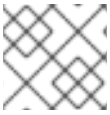
警告

SolutionFileIO 実装はスレッドセーフである必要があります。

14.2.3.4. データベース (またはその他のリポジトリー) からの入力ソリューションの読み込み

ベンチマーク設定は、現在データセット毎に **<inputSolutionFile>** 要素を指定する必要があります。データセットが、データベースまたは別のタイプのリポジトリーにある場合にこの問題に対処する方法は 2 つあります。

- データベースからデータセットを抽出し、(たとえば **XStreamSolutionFileIO** を使用した XML として) ローカルファイルにシリアル化します。次に、そのファイルを **<inputSolutionFile>** 要素として使用します。
- 各データセットに対して、データセットの固有の ID を保持するテキストファイルを作成します。その識別子を読み込み、その ID でデータベースに接続し、問題を抽出する カスタムの SolutionFileIO を記述します。そのテキストファイルは **<inputSolutionFile>** 要素として設定します。



注記

ローカルファイルの方が常に速く、ネットワーク接続も必須ではありません。

14.2.4. HotSpot コンパイラーのウォーミングアップ

HotSpot JIT コンパイル (そして、おそらく DRL コンパイルでも) CPU 時間がなくなるため、ウォーミングアップが終わっていない、最初 (もしくは最初の数回) のベンチマークは信頼できません。

このようなねじれを避けるために、ベンチマーカーでは、実際のベンチマークを実行する前に、複数のベンチマークを一定期間実行できます。通常、ウォーミングアップは 30 秒行えば十分となります。

```
<plannerBenchmark>
  ...
  <warmUpSecondsSpentLimit>30</warmUpSecondsSpentLimit>
  ...
</plannerBenchmark>
```

注記

このウォーミングアップに費やした時間は、データセットのロードにかかった時間には含まれません。データセットが大きくなると、設定に指定した時間よりもはるかに長い間、ウォーミングアップの実行にかかるためです。

14.2.5. ベンチマークの詳細: 事前定義済み設定

一般的な Solver 設定に対するベンチマークを短時間で設定する場合は、**solverBenchmarks** ではなく **solverBenchmarkBluePrint** を使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens</benchmarkDirectory>
  <warmUpSecondsSpentLimit>30</warmUpSecondsSpentLimit>

  <inheritedSolverBenchmark>
    <problemBenchmarks>

<xStreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</xStreamAnnotated
Class>
      <inputSolutionFile>data/nqueens/unsolved/32queens.xml</inputSolutionFile>
      <inputSolutionFile>data/nqueens/unsolved/64queens.xml</inputSolutionFile>
      <problemStatisticType>BEST_SCORE</problemStatisticType>
    </problemBenchmarks>
    <solver>
      <scanAnnotatedClasses/>
      <scoreDirectorFactory>
        <scoreDefinitionType>SIMPLE</scoreDefinitionType>
        <scoreDrl>org/optaplanner/examples/nqueens/solver/nQueensScoreRules.drl</scoreDrl>
        <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
      </scoreDirectorFactory>
      <termination>
        <minutesSpentLimit>1</minutesSpentLimit>
      </termination>
    </solver>
  </inheritedSolverBenchmark>

  <solverBenchmarkBluePrint>

<solverBenchmarkBluePrintType>EVERY_CONSTRUCTION_HEURISTIC_TYPE_WITH_EVERY_L
OCAL_SEARCH_TYPE</solverBenchmarkBluePrintType>
  </solverBenchmarkBluePrint>
</plannerBenchmark>
```

次の **SolverBenchmarkBluePrintTypes** に対応します。

- **EVERY_CONSTRUCTION_HEURISTIC_TYPE**: すべてのタイプの構築ヒューリスティック (FF、FFD、CI など) を実行します。
- **EVERY_LOCAL_SEARCH_TYPE**: デフォルトの構築ヒューリスティックを使用して、すべての局所選択タイプ (タブー選択、レイトアクセプランスなど) を実行します。
- **EVERY_CONSTRUCTION_HEURISTIC_TYPE_WITH_EVERY_LOCAL_SEARCH_TYPE**: すべての局所探索タイプで、すべての構築ヒューリスティックタイプを実行します。

14.2.6. ベンチマークを実行した出力ソリューションの記述

各ベンチマークを実行したときの最適解を **benchmarkDirectory** に記述できます。このファイルが使われることはほぼなく、サイズが肥大化すると考えられるので、これはデフォルトでは無効になっています。また、データセットが大きい時に 1 回のベンチマークでの最適解を記述すると、特に XStream

XML などの詳細なフォーマットでは、時間とメモリーを大きく消費 (し **OutOfMemoryError** が発生) する可能性があります。

benchmarkDirectory でこのような解 (ソリューション) を記述するには、**writeOutputSolutionEnabled** を有効にします。

```
<problemBenchmarks>
  ...
  <writeOutputSolutionEnabled>true</writeOutputSolutionEnabled>
  ...
</problemBenchmarks>
```

14.2.7. ベンチマークのログ

ベンチマークのログは、Solver ログ と同じように設定します。

ログメッセージを、ベンチマークごとに異なるファイルに分けるには、ShiftingAppender の MDC の **singleBenchmark.name** キーを使用します。たとえば、 Logback の場合は **logback.xml** に以下のように設定します。

```
<appender name="fileAppender" class="ch.qos.logback.classic.sift.SiftingAppender">
  <discriminator>
    <key>singleBenchmark.name</key>
    <defaultValue>app</defaultValue>
  </discriminator>
  <sift>
    <appender name="fileAppender.${singleBenchmark.name}" class="...FileAppender">
      <file>local/log/optaplannerBenchmark-${singleBenchmark.name}.log</file>
      ...
    </appender>
  </sift>
</appender>
```

14.3. ベンチマークレポート

14.3.1. HTML レポート

ベンチマークを実行すると、**benchmarkDirectory** に **index.html** という名前で HTML レポートが作成されます。このファイルをブラウザーで開きます。このファイルには、以下の内容を含むベンチマークの概要が記述されます。

- サマリー統計: グラフおよび表
- **inputSolutionFile** に関する問題の統計データ: グラフおよび CSV
- 各 Solver 設定 (ランク付け): コピーアンドペーストに便利
- ベンチマーク情報: 設定、ハードウェアなど

注記

グラフ生成には、優れた JFreeChart ライブラリーが使用されます。

HTML レポートの数値の書式には、デフォルトロケールが適用されます。作成したベンチマークレポートを他国の人と共有する場合は、**locale** を上書きすることも検討してください。

```
<plannerBenchmark>
  ...
  <benchmarkReport>
    <locale>en_US</locale>
  </benchmarkReport>
  ...
</plannerBenchmark>
```

14.3.2. Solver のランク付け

ベンチマークレポートでは、自動的に Solver にランクが付けられます。ランク **0** の **Solver** は、お気に入りの **Solver** と呼ばれます。総合的には最良のパフォーマンスですが、すべての問題で最良とは限りません。本番環境では、このお気に入りの **Solver** を使用することが推奨されます。

Solver のランク付け方法は複数あり、以下のように設定します。

```
<plannerBenchmark>
  ...
  <benchmarkReport>
    <solverRankingType>TOTAL_SCORE</solverRankingType>
  </benchmarkReport>
  ...
</plannerBenchmark>
```

以下の **solverRankingTypes** に対応しています。

- **TOTAL_SCORE** (デフォルト): 全体のスコアを最大にし、すべてのソリューションを実行した場合のコスト全体を最小限に抑えます。
- **WORST_SCORE**: 最悪のシナリオを最小限に抑えます。
- **TOTAL_RANKING**: 全体のランク付けを最大にします。これを使用すると、データセットのサイズまたは難易度が大きく異なる場合に、**Score** の差異を大きくします。

Solvers に失敗したベンチマークが 1 つでもあれば、ランク付けされません。完全に初期化されていないソリューションがある **Solvers** のランクは低くなります。

Comparator を実装すれば、カスタムのランクも使用できます。

```
  <benchmarkReport>

<solverRankingComparatorClass>...TotalScoreSolverRankingComparator</solverRankingComparatorClass>
  </benchmarkReport>
```

もしくは、**SolverRankingWeightFactory** を実装します。

```
  <benchmarkReport>

<solverRankingWeightFactoryClass>...TotalRankSolverRankingWeightFactory</solverRankingWeightFactoryClass>
  </benchmarkReport>
```

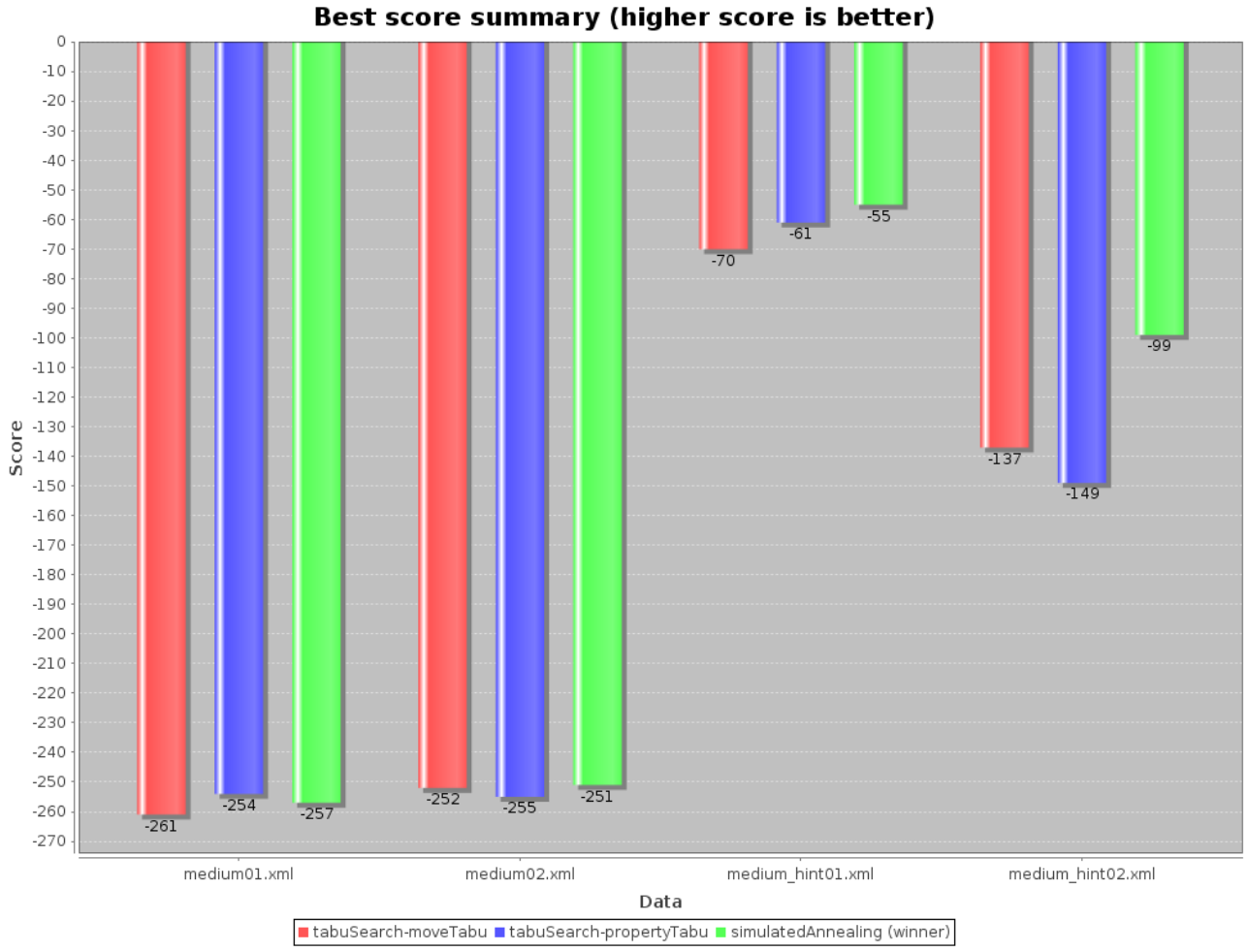
14.4. 要約統計

14.4.1. 最高スコアの要約 (グラフおよび表)

各 Solver の **inputSolutionFile** に対する最高スコアを表示します。

最適な Solver 設定を可視化するのに便利です。

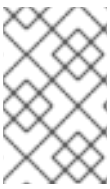
図14.1 最高スコアの要約統計



14.4.2. 最高スコアのスケーラビリティー要約 (グラフ)

各 Solver 設定の問題スケールあたりの最高スコアを表示します。

各 Solver 設定のスケーラビリティーを可視化するのに便利です。



注記

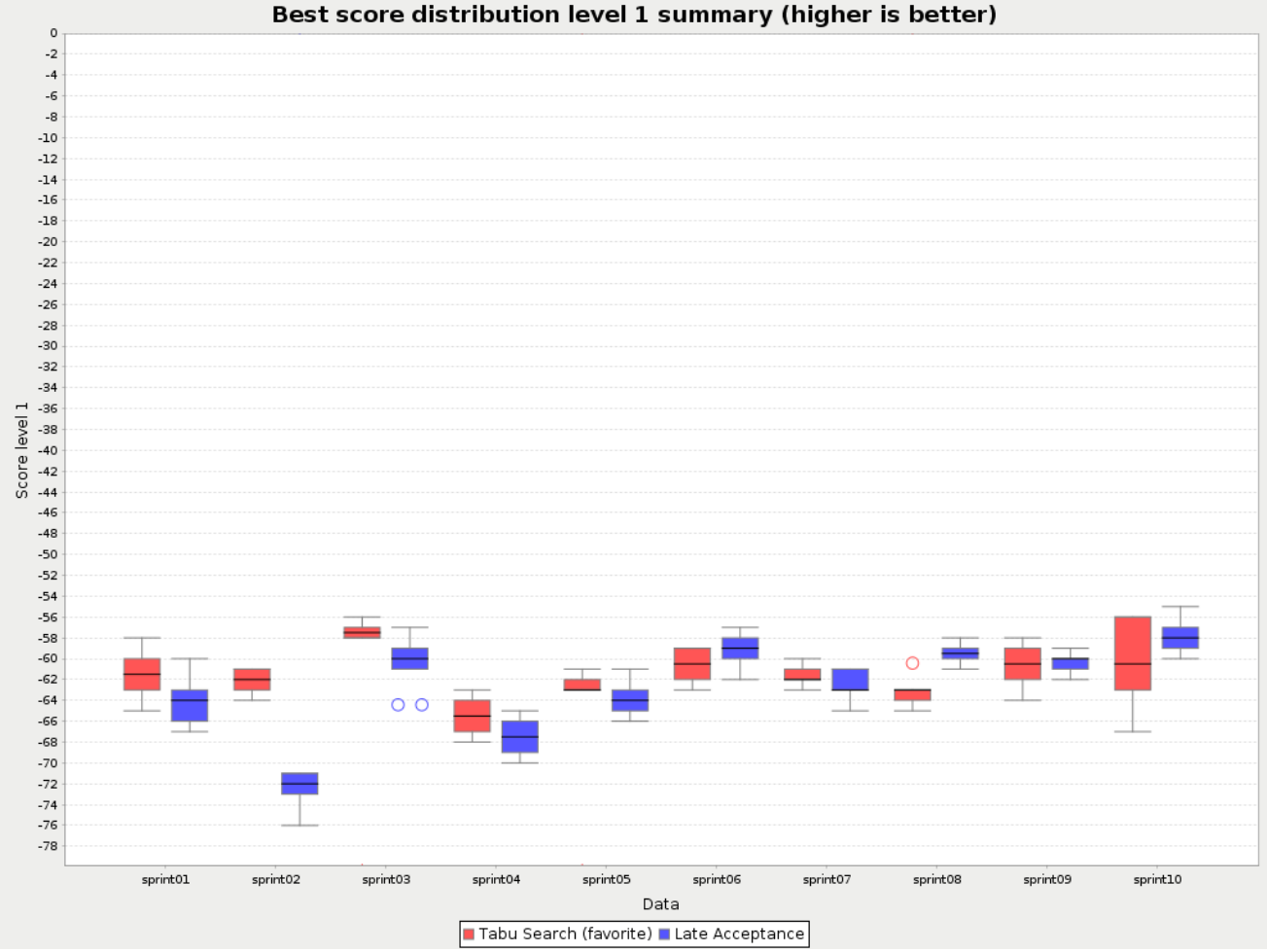
@ValueRangeProvider メソッドのシグネチャーが、(**CountableValueRange** または **Collection** ではなく) **ValueRange** を返すと、問題のスケールが **0** を報告します。相違点については **ValueRangeFactory** を参照してください。

14.4.3. 最高スコア分布の要約 (グラフ)

各 Solver 設定の **inputSolutionFile** に従って、最高スコアを表示します。

各 Solver 設定の信頼性を可視化するのに役に立ちます。

図14.2 最高スコア分布の要約統計



この要約を使用するには、統計ベンチマーク を有効にします。

14.4.4. 勝利スコアの差の要約 (グラフおよび表)

各 Solver 設定の **inputSolutionFile** に従って、勝利スコアの差を表示します。勝利スコアの差は、その特定の **inputSolutionFile** と勝利 Solver 設定のスコアを比較したスコアの差となります。

最高スコア要約の結果を拡大するのに役に立ちます。

14.4.5. 最低スコアの差の割合 (ROI) の要約 (グラフおよび表)

各 Solver 設定の **inputSolutionFile** に従って、投資効果率 (ROI) を表示します (この特定の **inputSolutionFile** に対する最低の Solver 設定からアップグレードする場合)。

意思決定者への投資対効果を可視化するのに便利です。

14.4.6. 平均計算数の要約 (グラフおよび表)

スコア計算の速度 (各 Solver 設定に対する問題スケールあたりの平均計算数) を表示します。

異なるスコア計算プログラムや、スコアルール実装を比較するのに便利です (Solver 設定は同じであることが前提です)。また、追加の制約を行った場合のスケーラビリティーのコストを測定するのにも役に立ちます。

14.4.7. 経過時間の要約 (グラフおよび表)

各 Solver 設定の **inputSolutionFile** に従って、経過時間を表示します。固定時間制限に対してベンチマークを設定している場合、これは有益ではありません。

構築ヒューリスティックのパフォーマンスを可視化するのに便利です (その他の Solver フェーズが設定されていないことを前提としています)。

14.4.8. スケーラビリティーにかかった時間の要約 (グラフ)

各 Solver 設定の問題スケールにかかった時間を表示します。固定時間制限に対してベンチマークを行っている場合、これは有益ではありません。

構築ヒューリスティックのスケーラビリティーを推定するのに便利です (その他の Solver フェーズが設定されていないことを前提としています)。

14.4.9. 経過時間ごとの最高スコアの要約 (グラフ)

各 Solver 設定における経過時間ごとの最高スコアを表示します。固定時間制限に対してベンチマークを行っている場合、これは有益ではありません。

構築ヒューリスティックで、最高スコアと経過時間のトレードオフを可視化するのに便利です (その他の Solver フェーズが設定されていないことを前提としています)。

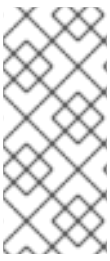
14.5. データセットに関する統計 (グラフおよび CSV)

14.5.1. 問題の統計の有効化

ベンチマーカーは、問題の統計をグラフおよび CSV (コンマ区切り値) ファイルとして **benchmarkDirectory** に出力するのをサポートします。これを設定するには、**problemStatisticType** 行を追加します。

```
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens/solved</benchmarkDirectory>
  <inheritedSolverBenchmark>
    <problemBenchmarks>
      ...
      <problemStatisticType>BEST_SCORE</problemStatisticType>
      <problemStatisticType>CALCULATE_COUNT_PER_SECOND</problemStatisticType>
    </problemBenchmarks>
    ...
  </inheritedSolverBenchmark>
  ...
</plannerBenchmark>
```

複数の **problemStatisticType** 要素が許可されます。



注記

データセットごとの統計は、Solver の速度を著しく落とし、ベンチマークの結果に影響する可能性があります。したがって、この設定は任意で、デフォルトでは有効ではありません。

必須の要約統計が、Solver の速度を著しく落とすことはありません。

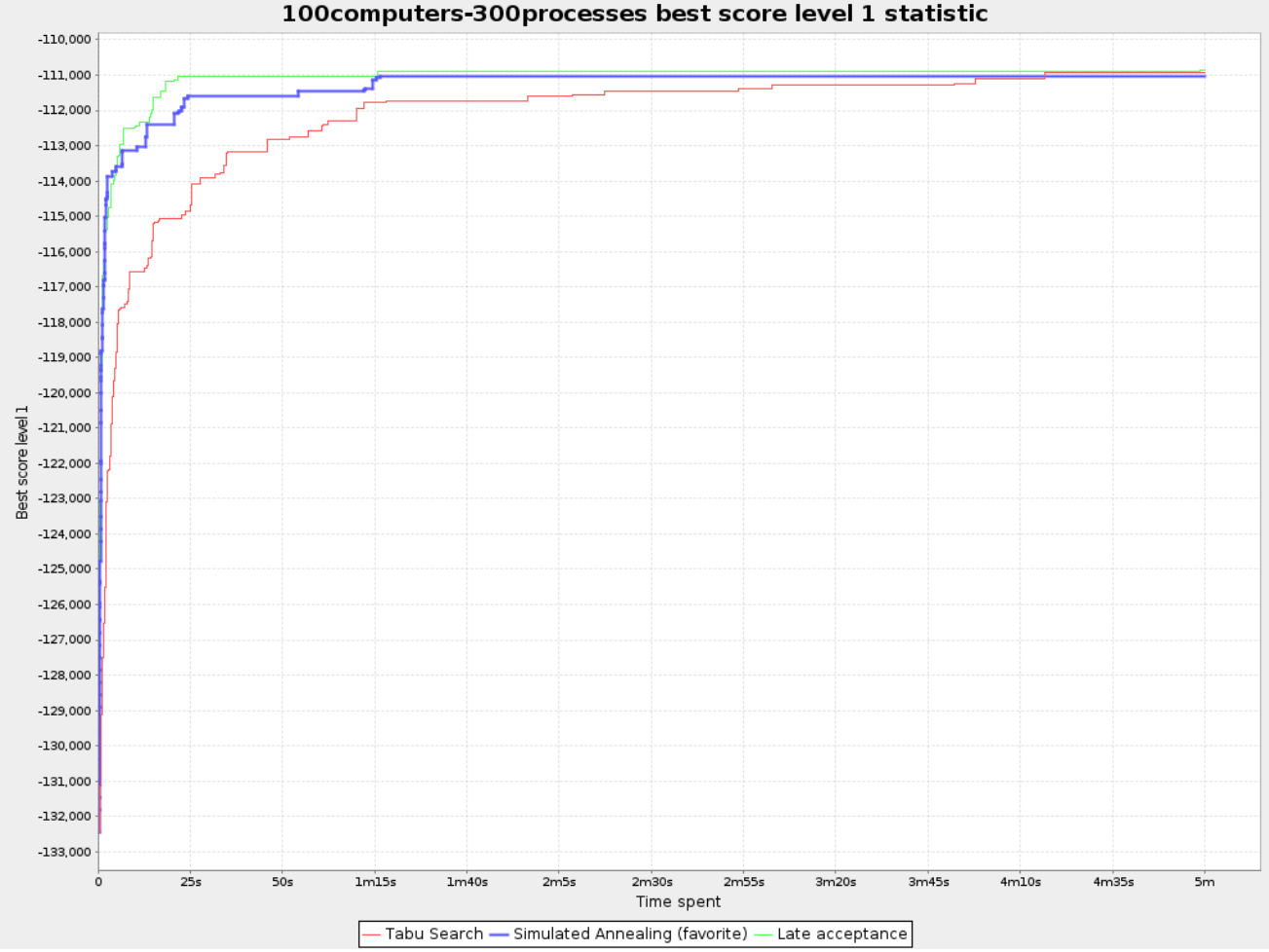
以下のタイプがサポートされます。

14.5.2. 経時最高スコア統計 (グラフおよび CSV)

時間の経過とともに最高スコアがどのように進化していったかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <problemStatisticType>BEST_SCORE</problemStatisticType>
</problemBenchmarks>
```

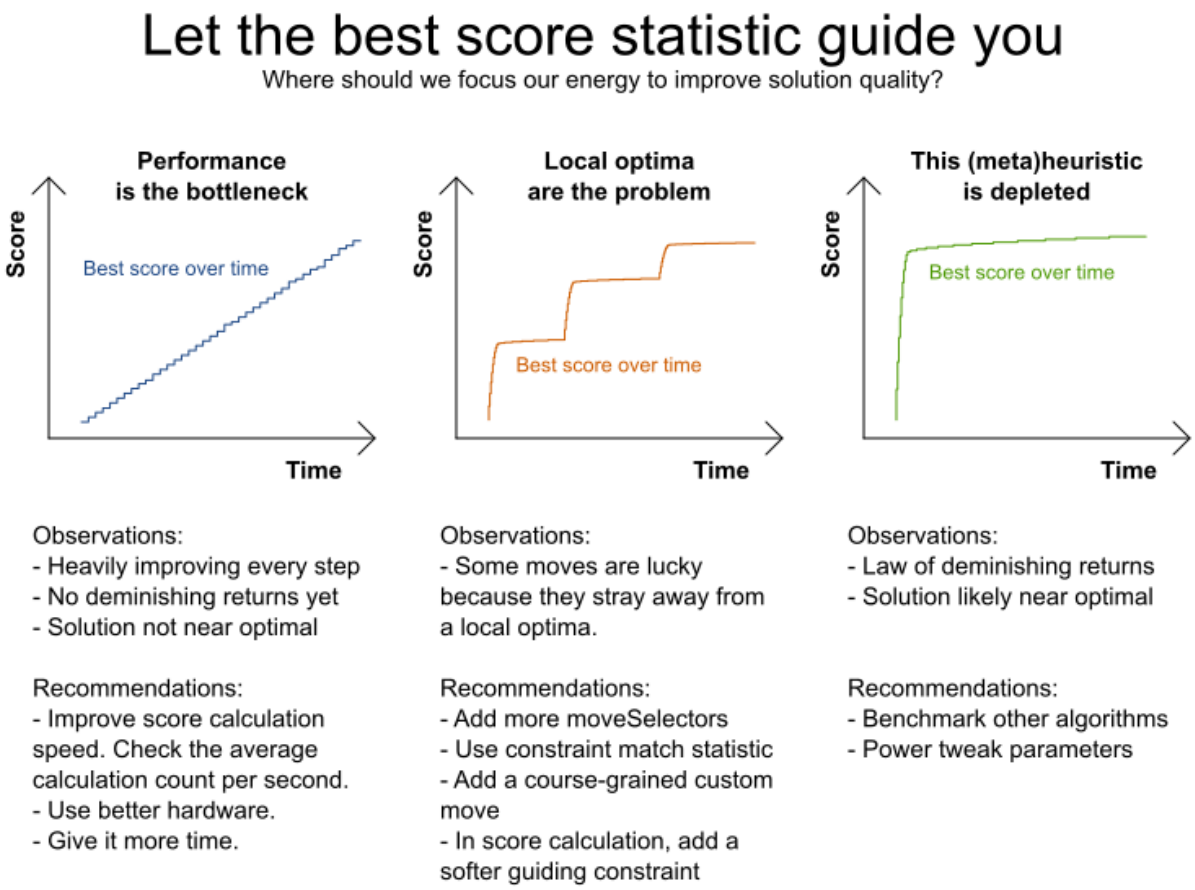
図14.3 経時最高スコア統計



注記

(焼きなまし法などの) 時間勾配ベースのアルゴリズムは、実行する時間制限設定を変更すると、統計が変わります。これは、この焼きなましの実装が、経過した時間に基づいて自動的に速度を決定するためです。一方、タブー探索およびレイトアクセプタンスについては、自動的に変更することはありません。

経時最高スコア統計は、Solver が局所最適条件で一時的に身動きできなくなる スコアトラップ などの潜在的な異常を検出するのに非常に役に立ちます。

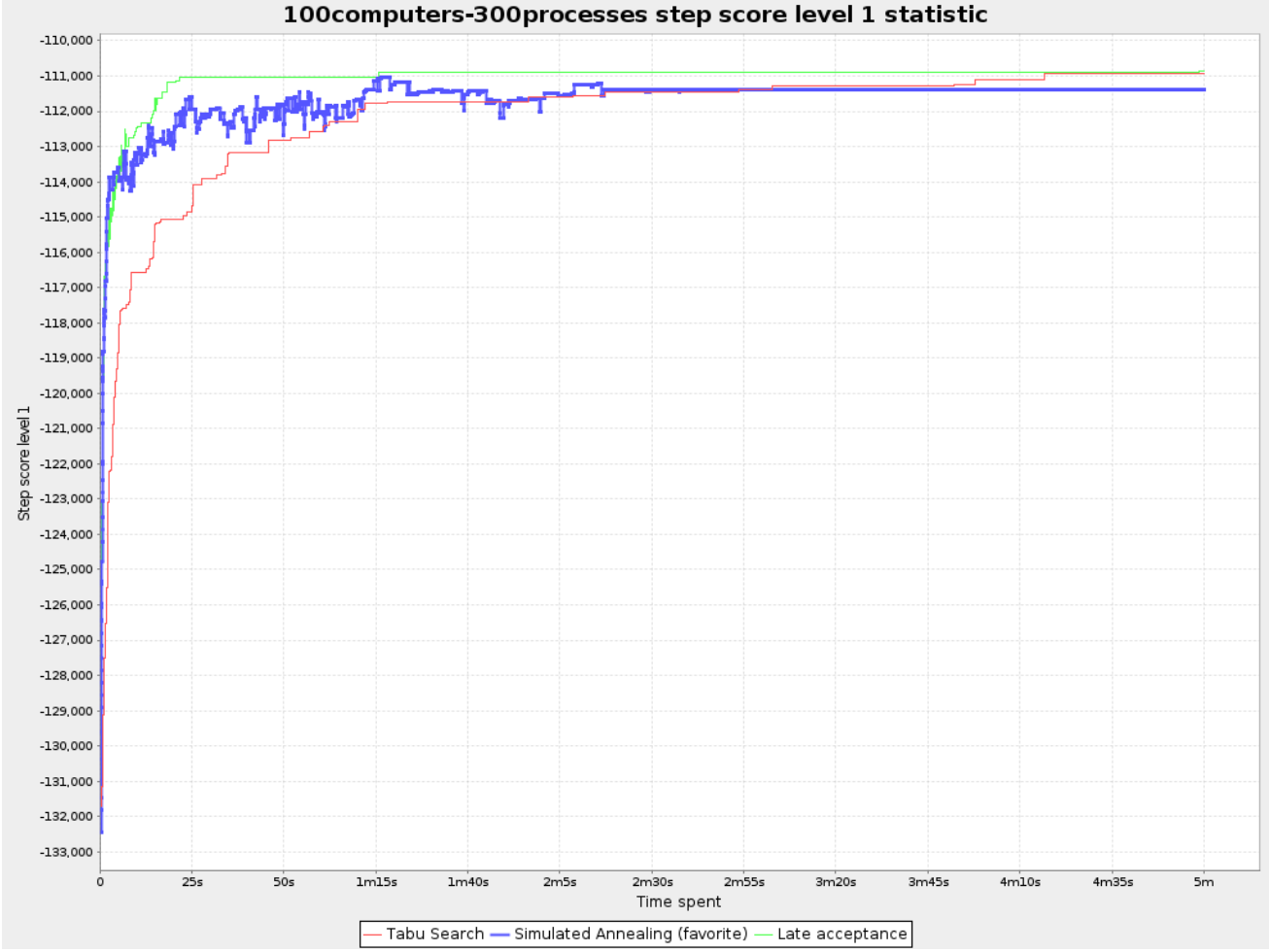


14.5.3. 経時ステップスコア統計 (グラフおよび CSV)

時間の経過とともにステップスコアがどのように進化していったかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <problemStatisticType>STEP_SCORE</problemStatisticType>
</problemBenchmarks>
```

図14.4 経時ステップスコア統計



ステップスコア統計と、最高スコア統計を比較します (特に最高スコアが水平になる部分)。それが局所最適条件に到達した場合、Solver は、悪化したステップを選択して回避する必要があります。ただし、あまり悪化させることはできません。

警告

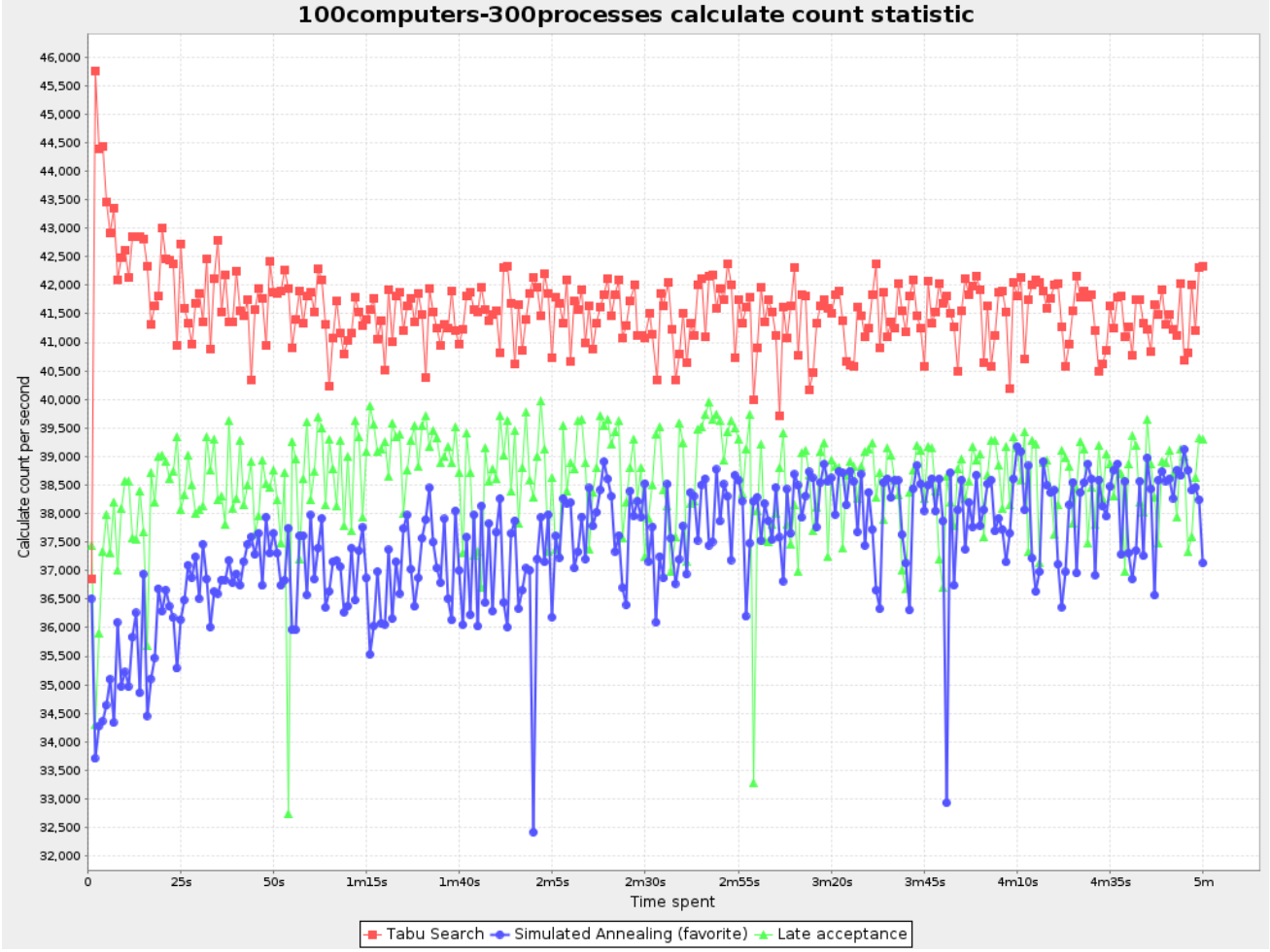
ステップスコア統計では、特に (焼きなまし法およびレイトアクセプタンスなどの) ステップが速いアルゴリズムにおいて、GC ストレスが原因で Solver が著しく遅くなっているのが確認できます。

14.5.4. 秒あたりカウントの計算統計 (グラフおよび CSV)

スコアが計算される速度を確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <problemStatisticType>CALCULATE_COUNT_PER_SECOND</problemStatisticType>
</problemBenchmarks>
```

図14.5 秒あたりカウントの計算統計



注記

ソリューションの初期化時に初期計算のカウントが高くなるのは一般的です。ごく一部のプランニングエンティティーだけを初期化している時点でのソリューションのスコア計算は、すべてのプランニングエンティティーを初期化したときよりはるかに簡単になります。

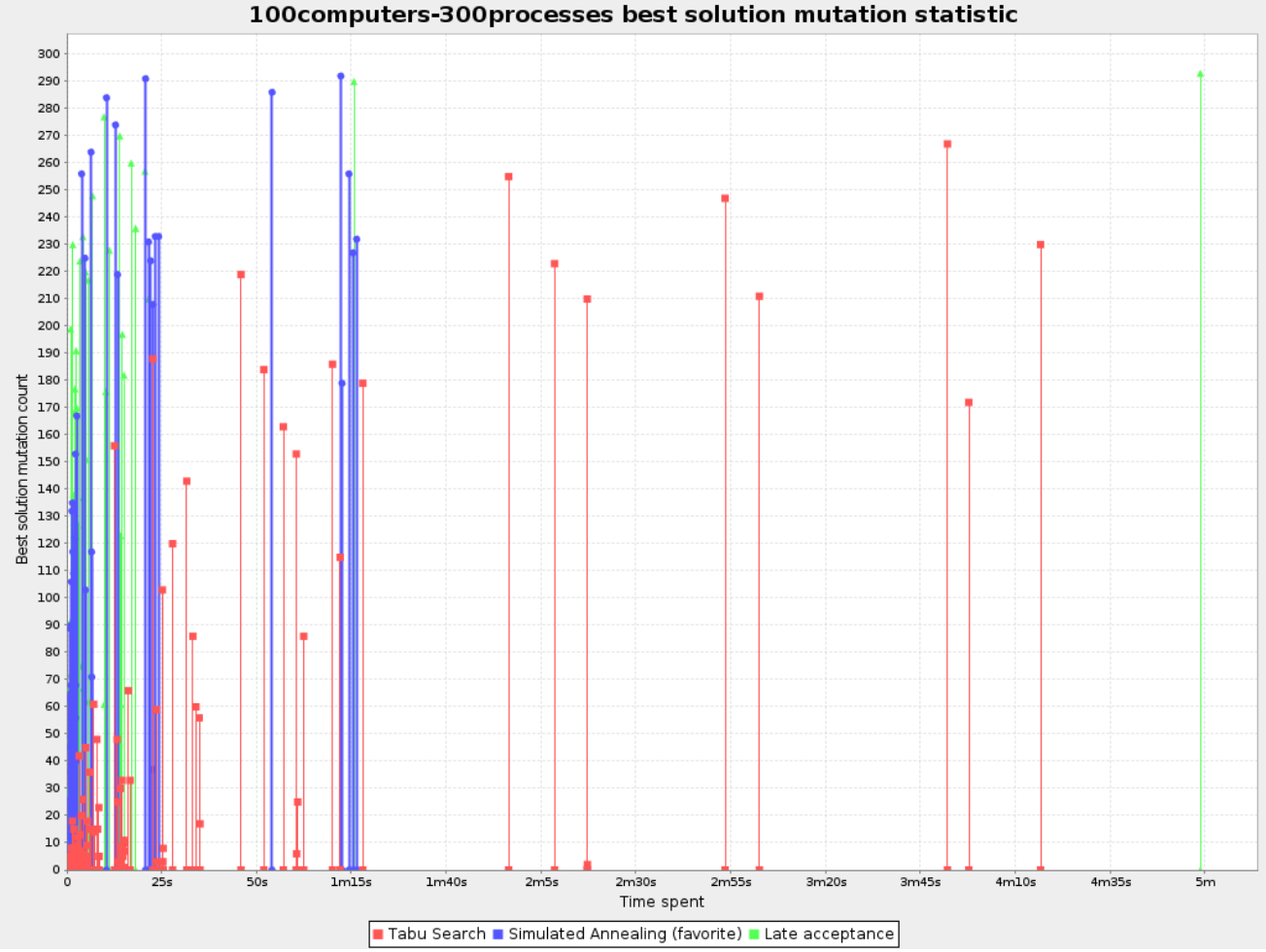
初期化を数秒間行ったあと、時折発生するストップザワールドのガベージコレクションの分裂を除き、計算カウントは比較的安定します。

14.5.5. 経時最適解変化統計 (グラフおよび CSV)

新しい最適解が 1 つ前の最適解 とどのぐらい異なるかを確認するには、以下を追加します。ここでは、値が異なるプランニング変数の数を数えていますが、複数回変更して最後には同じ値になった変数は除外されます。

```
<problemBenchmarks>
  ...
  <problemStatisticType>BEST_SOLUTION_MUTATION</problemStatisticType>
</problemBenchmarks>
```

図14.6 経時最適解変化統計



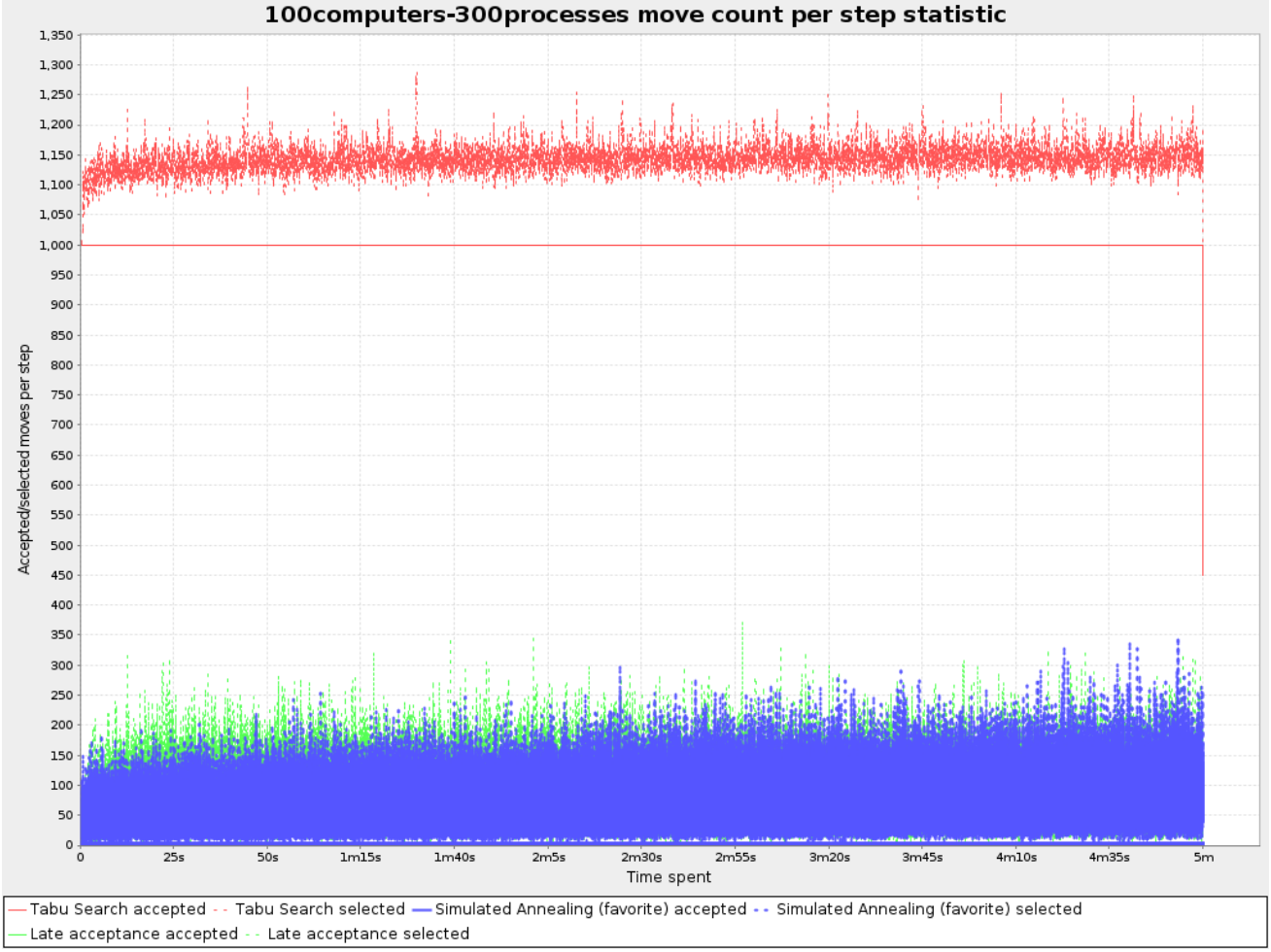
(人間のように振舞うアルゴリズムである) タブー探索を使用して、人間が、1 つ前の最適解を改善して新しい最適解にするのがどのぐらい難しいか判断します。

14.5.6. ステップあたりの Move 数 (グラフおよび CSV)

各ステップで選択して許可した Move 数が、時間の経過とともにどのように進化していったかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <problemStatisticType>MOVE_COUNT_PER_STEP</problemStatisticType>
</problemBenchmarks>
```

図14.7 ステップ値の Move 数



警告

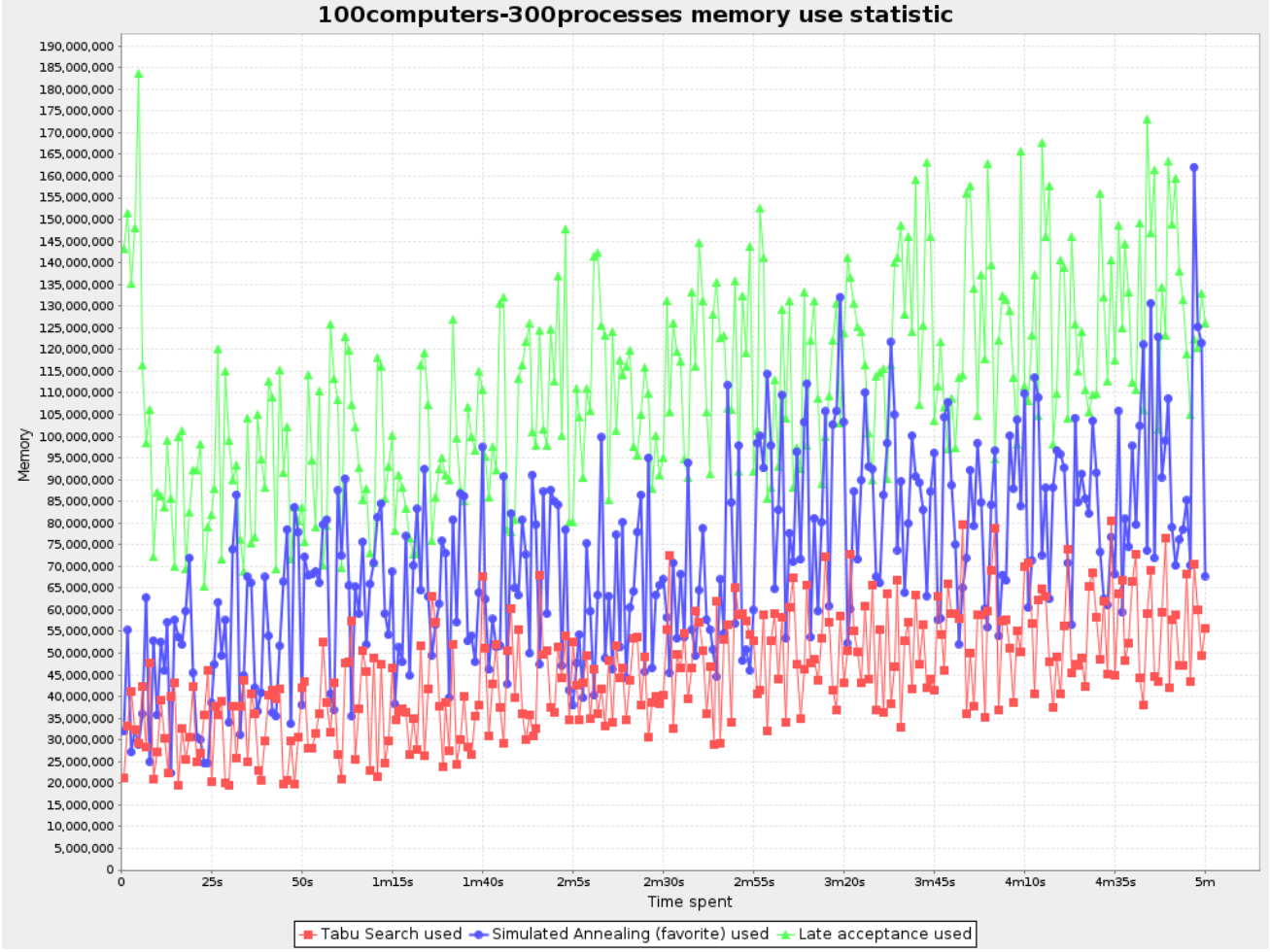
この統計では、特に (焼きなまし法、レイトアクセプタンスなどの) ステップの速いアルゴリズムにおいて、GC ストレスが原因で Solver が著しく遅くなっているのが確認できます。

14.5.7. メモリー使用統計 (グラフおよび CSV)

メモリーの使用量を確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <problemStatisticType>MEMORY_USE</problemStatisticType>
</problemBenchmarks>
```

図14.8 メモリー使用統計



警告

メモリー使用統計は、Solver に大きく影響することが確認されています。

14.6. ベンチマークに関する統計 (グラフおよび CSV)

14.6.1. 単一統計を有効にする

単一統計は、1 つの Solver 設定に対する 1 つのデータセットに関する統計です。問題の統計とは異なり、Solver 設定から集計しません。

ベンチマーカーは、1 つ統計を グラフおよび CSV (コンマ区切り値) ファイルとして **benchmarkDirectory** に出力するのをサポートします。設定する場合は **singleStatisticType** 行を追加します。

```
<plannerBenchmark>
  <benchmarkDirectory>local/data/nqueens/solved</benchmarkDirectory>
  <inheritedSolverBenchmark>
    <problemBenchmarks>
      ...
```

```
    <problemStatisticType>...</problemStatisticType>
    <singleStatisticType>PICKED_MOVE_TYPE_BEST_SCORE_DIFF</singleStatisticType>
  </problemBenchmarks>
  ...
</inheritedSolverBenchmark>
...
</plannerBenchmark>
```

複数の **singleStatisticType** 要素が許可されます。

注記

ベンチマークあたりの統計は、Solver の速度を著しく落とし、ベンチマークの結果に影響する可能性があります。したがって、この設定は任意で、デフォルトでは有効ではありません。

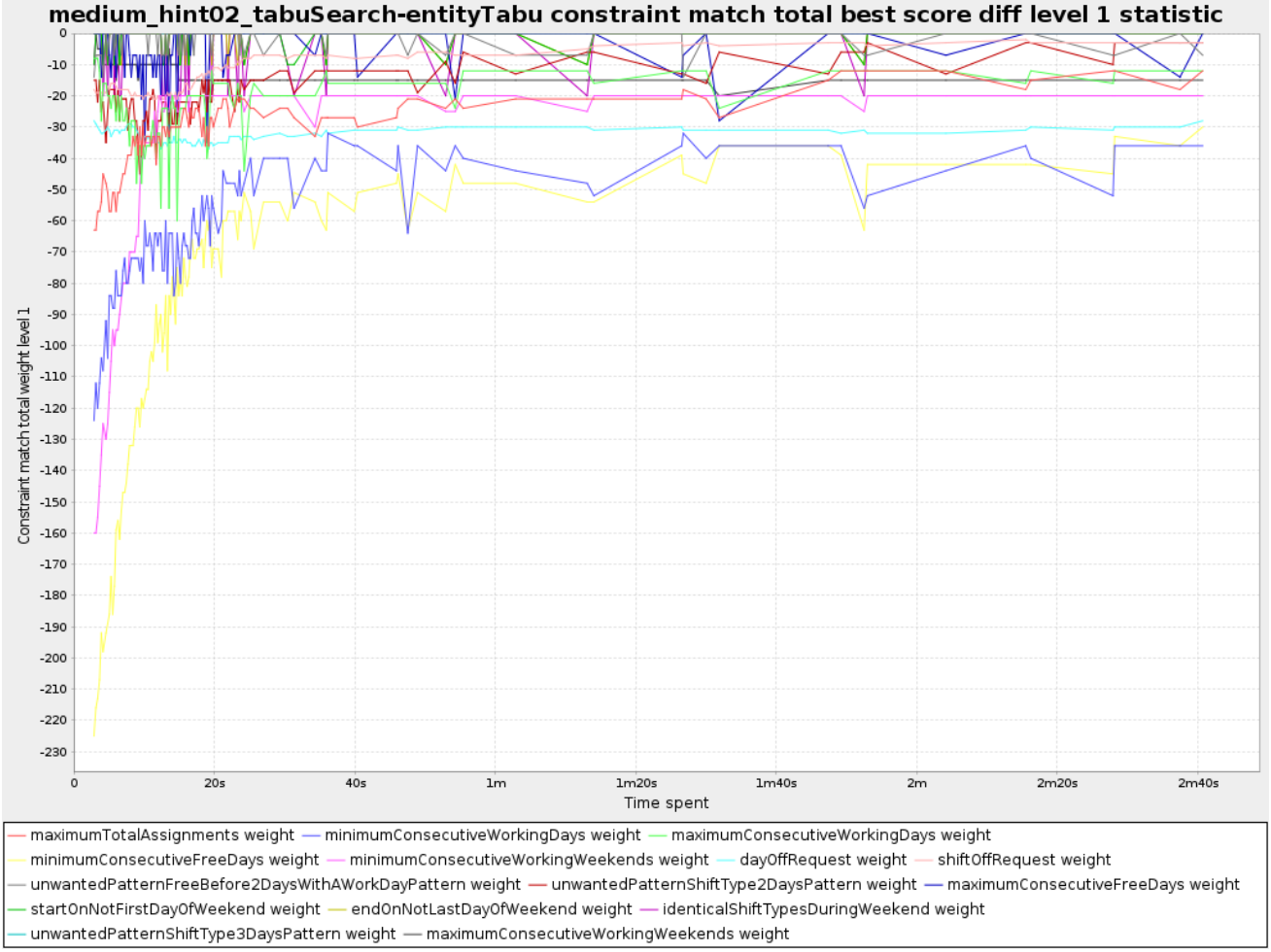
以下のタイプがサポートされます。

14.6.2. 経時最高スコアの合計に一致する制約統計 (グラフおよび CSV)

どの制約が経時最高スコアに (どのぐらい) 一致するかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <singleStatisticType>CONSTRAINT_MATCH_TOTAL_BEST_SCORE</singleStatisticType>
</problemBenchmarks>
```

図14.9 経時最高スコアの合計に一致する制約統計



スコア計算が、制約の一致 をサポートする必要があります。Drools スコア計算 では、制約の一致が自動的にサポートされますが、Java インクリメント演算子によるスコア計算 ではさらなる作業が必要です。

警告

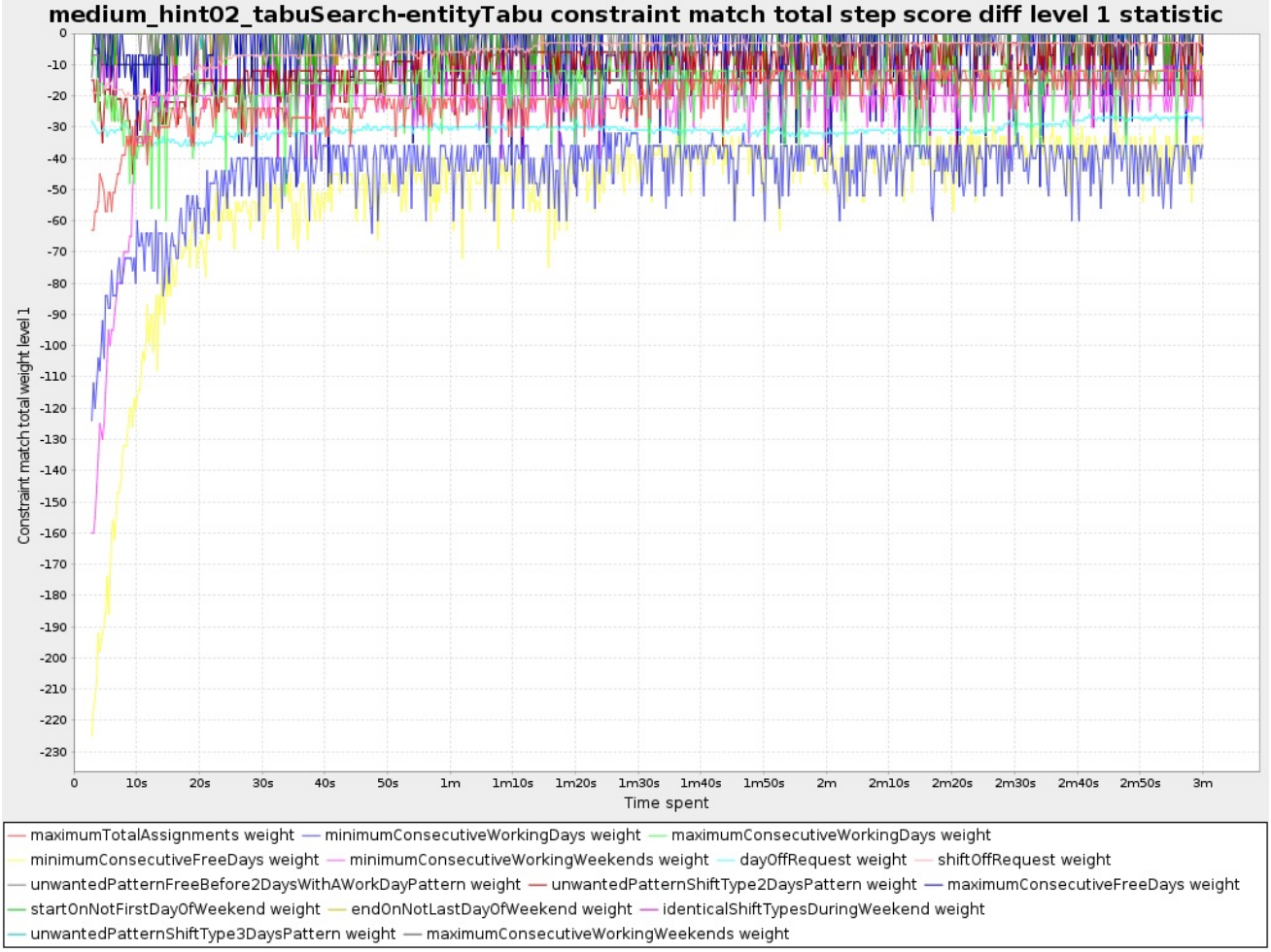
制約一致の合計統計は、Solver に大きく影響することが確認されています。

14.6.3. 経時ステップスコア合計に一致する制約統計 (グラフおよび CSV)

どの制約が、時間の経過とともにステップスコアに (どのぐらい) 一致するかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <singleStatisticType>CONSTRAINT_MATCH_TOTAL_STEP_SCORE</singleStatisticType>
</problemBenchmarks>
```

図14.10 経時ステップスコアの合計に一致する制約統計



スコア計算が、制約の一致 をサポートする必要があります。Drools スコア計算 では、制約の一致が自動的にサポートされますが、Java インクリメント演算子によるスコア計算 ではさらなる作業が必要です。

警告

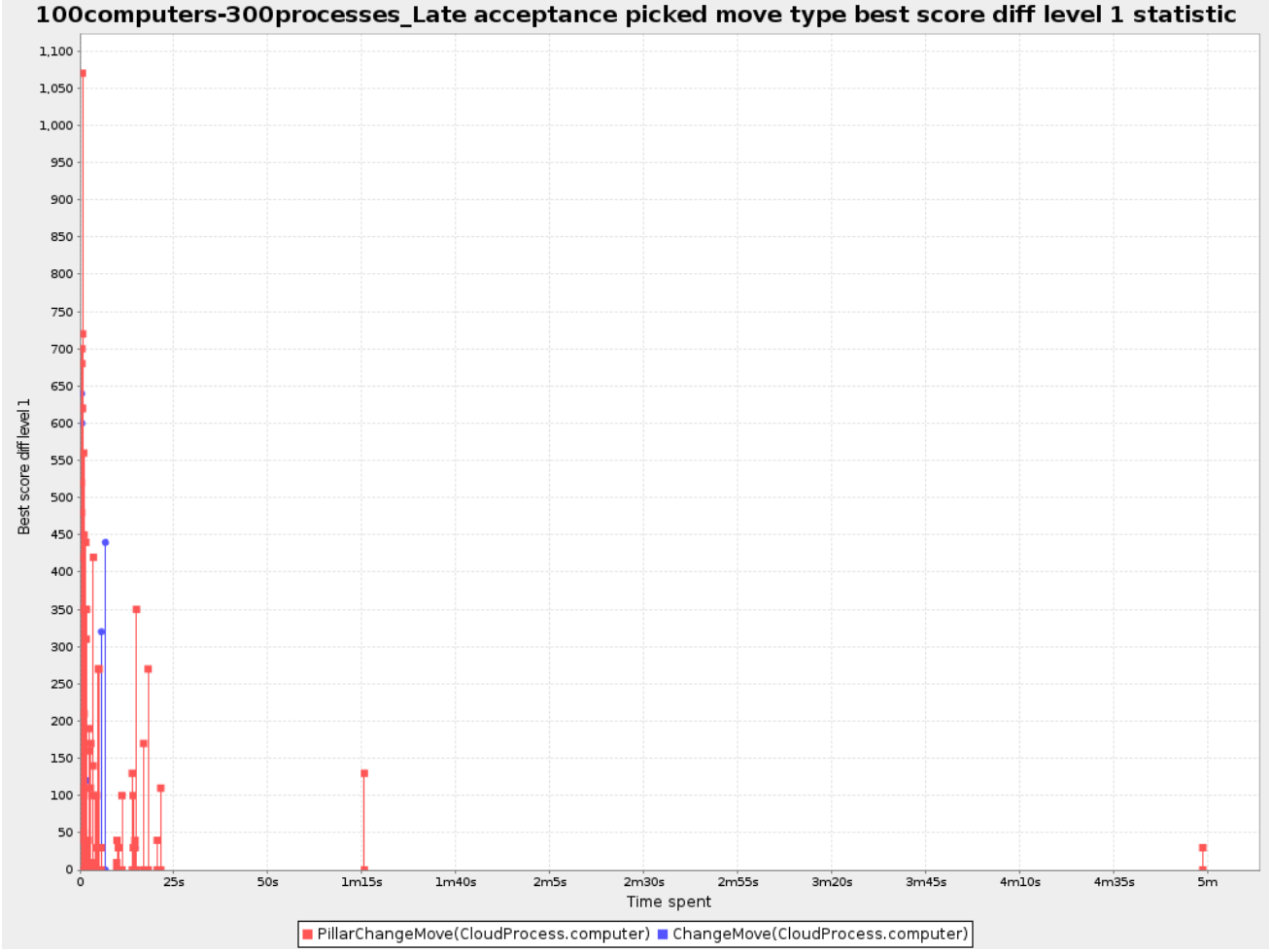
制約一致の合計統計は、Solver に大きく影響することが確認されています。

14.6.4. 選択した Move タイプの経時最高スコアの開き統計 (グラフおよび CSV)

どの Move タイプが、時間の経過とともに最高スコアが (どのように) 改善したかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <singleStatisticType>PICKED_MOVE_TYPE_BEST_SCORE_DIFF</singleStatisticType>
</problemBenchmarks>
```

図14.11 選択した Move タイプの経時最高スコアの開き統計

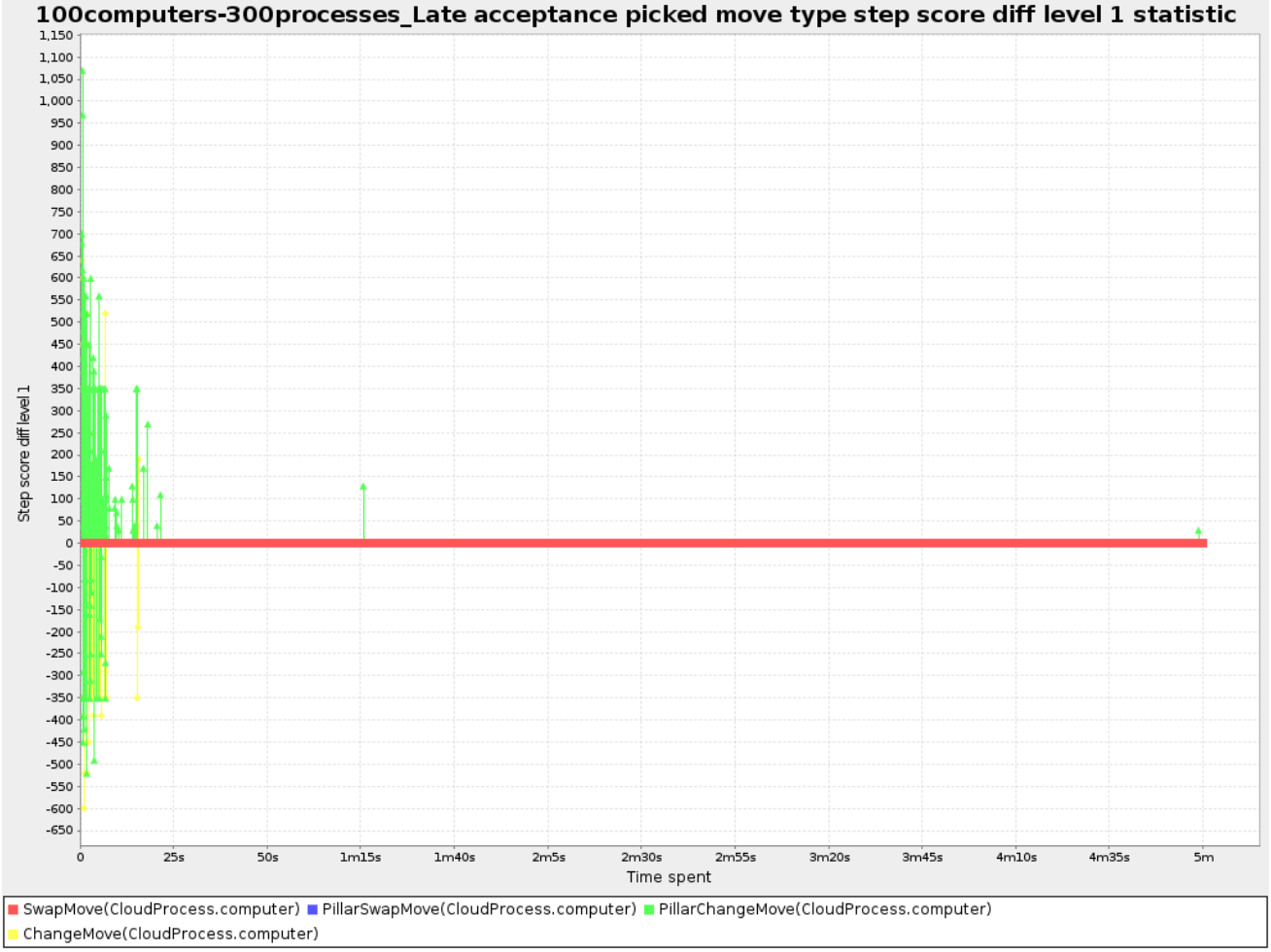


14.6.5. 選択した Move タイプの経時ステップスコアの開き統計 (グラフおよび CSV)

各勝利ステップが、時間の経過とともにステップスコアにどのぐらい影響するかを確認するには、以下を追加します。

```
<problemBenchmarks>
  ...
  <singleStatisticType>PICKED_MOVE_TYPE_STEP_SCORE_DIFF</singleStatisticType>
</problemBenchmarks>
```

図14.12 選択した Move タイプの経時ステップスコアの開き統計



14.7. 詳細なベンチマーク

14.7.1. ベンチマークのパフォーマンスに効果的な方法

14.7.1.1. 複数のスレッドで同時に行うベンチマーク

コンピューターで複数のプロセッサーが利用できる場合は、複数のスレッドで同時に複数のベンチマークを実行して、ベンチマークの結果をより速く取得することができます。

```
<plannerBenchmark>
  ...
  <parallelBenchmarkCount>AUTO</parallelBenchmarkCount>
  ...
</plannerBenchmark>
```

警告

一度に実行するベンチマークが多すぎると、ベンチマークの結果に悪影響を与えます。ガベージコレクションやその他のプロセスに使用するために、一部のプロセッサーは残すようにしてください。

parallelBenchmarkCount を **AUTO** にすれば、ベンチマーク結果の信頼性と効率が最大になります。

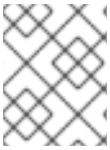
以下の **parallelBenchmarkCounts** がサポートされます。

- **1** (デフォルト): すべてのベンチマークを順次実行します。
- **AUTO**: 同時に実行するベンチマークの数を Planner が決定します。この計算式は経験に基づいています。その他の並行許可オプションよりも、このオプションを使用することが推奨されます。
- 統計数: 同時に実行するベンチマークの数。

```
<parallelBenchmarkCount>2</parallelBenchmarkCount>
```

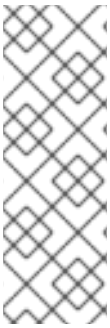
- JavaScript 計算式: 同時に実行するベンチマークの数を計算する式。**availableProcessorCount** 変数を使用できます。たとえば、以下のようになります。

```
<parallelBenchmarkCount>(availableProcessorCount / 2) + 1</parallelBenchmarkCount>
```



注記

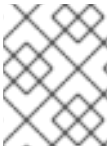
parallelBenchmarkCount は、常に利用可能なプロセッサー数に制限されます。その値よりも高くなると、自動的に減らされます。



注記

コンピューターの冷却が遅い、または信頼できない場合は、**parallelBenchmarkCount** を 1 以上 (もしくは **AUTO**) にすると、CPU がオーバーヒートする可能性があります。

sensors コマンドを使用すれば、オーバーヒートの可能性があることを検出できます。このコマンドは、ほとんどの Linux ディストリビューションでは **lm_sensors** パッケージまたは **lm-sensors** パッケージで利用できます。Windows にも無料ソフトが複数利用できます。



注記

今後、複数の JVM ベンチマークをサポートする予定です。この機能は、マルチスレッド解決 またはマルチ JVM 解決とは無関係です。

14.7.2. 統計ベンチマーク

環境と、ベンチマークの結果に対する乱数ジェネレーターへの影響を最小限に抑えるために、ベンチマークを 1 回実行する際に繰り返す回数を設定します。この実行の結果は統計的に集められます。個々の結果については 最高スコア分布の要約 が作成されますが、レポートでも確認できます。

<subSingleCount> 要素を **<inheritedSolverBenchmark>** 要素、または **<solverBenchmark>** 要素に追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<plannerBenchmark>
  ...
  <inheritedSolverBenchmark>
    ...
    <solver>
      ...
    </solver>
    <subSingleCount>10<subSingleCount>
  </inheritedSolverBenchmark>
  ...
</plannerBenchmark>
```

subSingleCount のデフォルトは **1** (統計ベンチマークなし) に設定されています。

注記

subSingleCount を **1** より大きくすると、**EnvironmentMode REPRODUCIBLE** 以下では、ベンチマーカーが (サブシングルインデックスに対して) 再現性を失わずに、サブシングルの実行に対して、自動的に異なる ランダムシード を使用します。

14.7.3. テンプレートベースのベンチマークと、マトリックスベンチマーク

マトリックスベンチマーキングは、ベンチマーク値を組み合わせて検査を行います。たとえば、ベンチマークの 4 つの **entityTabuSize** 値 (**5**、**7**、**11**、および **13**) と 3 つの **acceptedCountLimit** 値 (**500**、**1000**、および **2000**) を組み合わせると、Solver 設定は 12 個になります。

このようなベンチマーク設定の詳細レベルを減らすには、ベンチマークの設定に FreeMarker テンプレートを使用することもできます。

```
<plannerBenchmark>
  ...

  <inheritedSolverBenchmark>
    ...
  </inheritedSolverBenchmark>

<#list [5, 7, 11, 13] as entityTabuSize>
<#list [500, 1000, 2000] as acceptedCountLimit>
  <solverBenchmark>
    <name>entityTabuSize ${entityTabuSize} acceptedCountLimit ${acceptedCountLimit}</name>
    <solver>
      <localSearch>
        <unionMoveSelector>
          <changeMoveSelector/>
          <swapMoveSelector/>
        </unionMoveSelector>
        <acceptor>
```

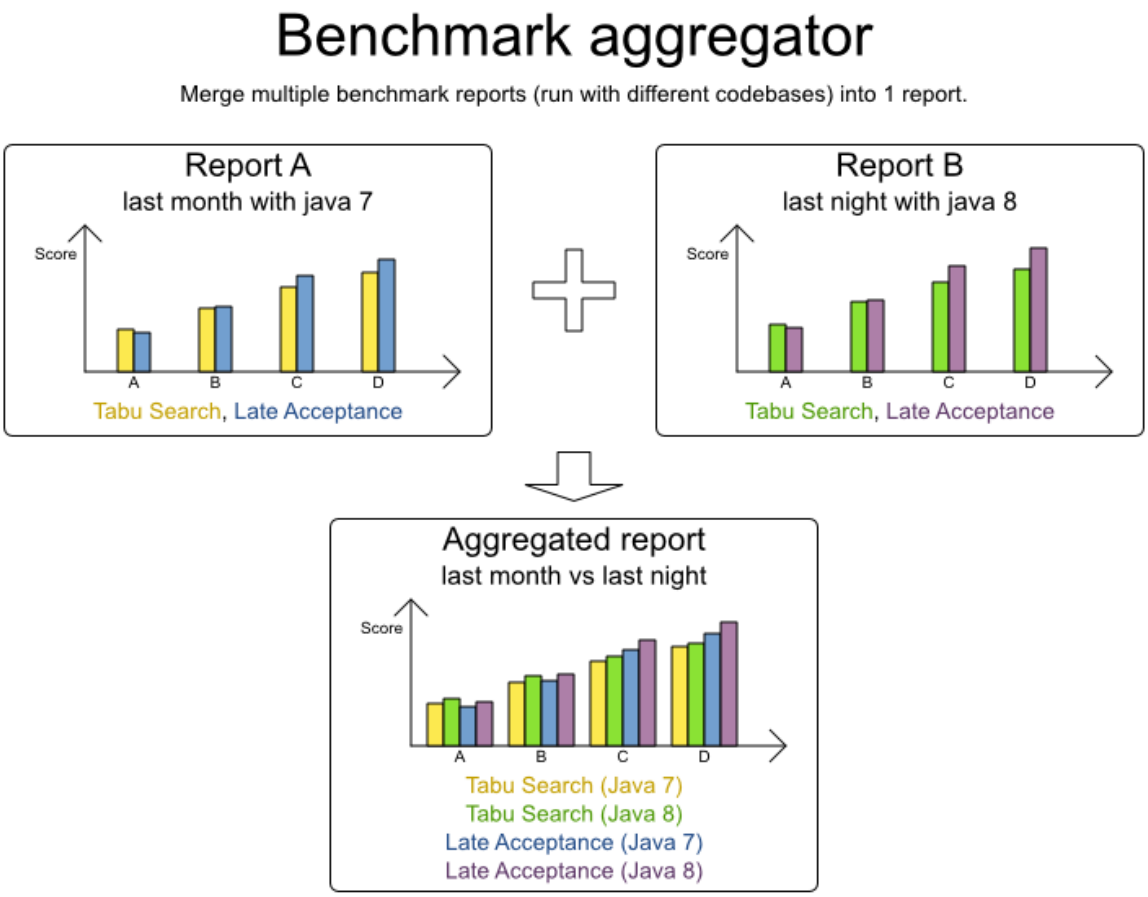
```
        <entityTabuSize>${entityTabuSize}</entityTabuSize>
      </acceptor>
      <forager>
        <acceptedCountLimit>${acceptedCountLimit}</acceptedCountLimit>
      </forager>
    </localSearch>
  </solver>
 </solverBenchmark>
</#list>
</#list>
</plannerBenchmark>
```

そして、**PlannerBenchmarkFactory** クラスでビルトします。

```
        PlannerBenchmarkFactory plannerBenchmarkFactory =
PlannerBenchmarkFactory.createFromFreemarkerXmlResource(

"org/optaplanner/examples/cloudbalancing/benchmark/cloudBalancingBenchmarkConfigTemplate.xml.ft
");
        PlannerBenchmark plannerBenchmark = plannerBenchmarkFactory.buildPlannerBenchmark();
```

14.7.4. ベンチマークレポート集約

BenchmarkAggregator は、1つ以上の既存ベンチマークをマージして新しいベンチマークレポートを作成します。ベンチマークが新たに実行することはありません。



これは、以下に役に立ちます。

- コード変更の影響をレポート: コード変更の前後で同じベンチマーク設定を実行して、レポートを統合します。
- 依存関係のアップグレードの影響をレポート: 依存関係のアップグレード前後で同じベンチマーク設定を実行して、レポートを統合します。
- 詳細なレポートを要約する: 既存レポートの中から、重要な Solver ベンチマークだけを選択します。これは、テンプレートレポートでグラフを読みやすくするのに役に立ちます。
- ベンチマークを一部再実行: 既存レポートの一部 (たとえば失敗したり無効になっていた Solver だけ) を再実行して、新しい値を使って当初意図したレポートを再作成します。

これを使用するには、**PlannerBenchmarkFactory** を **BenchmarkAggregatorFrame** に提供して、GUI を表示します。

```
    public static void main(String[] args) {
        PlannerBenchmarkFactory plannerBenchmarkFactory =
PlannerBenchmarkFactory.createFromXmlResource(
                "org/optaplanner/examples/nqueens/benchmark/nqueensBenchmarkConfig.xml");
        BenchmarkAggregatorFrame.createAndDisplay(plannerBenchmarkFactory);
    }
```

警告

ベンチマーク設定を入力として使用しているにもかかわらず、その設定で、**<benchmarkDirectory>** 要素と **<benchmarkReport>** 要素を除いたすべての要素を無視します。

GUI で、重要なベンチマークだけを選択して、ボタンを押してレポートを作成します。

注記

マージされるすべての入力レポートは、**BenchmarkAggregator** と同じ Planner バージョンで生成している必要があります (ホットフィックスは同じである必要はありません)。ベンチマークレポートのデータ構造は頻繁に変更するため、メジャーまたはマイナーのバージョンが異なる Planner で作成したレポートでは、正しい情報が提供されていることが保証されません。

第15章 反復計画

15.1. 反復計画の概要

世の中は常に変化しています。解 (ソリューション) を作成するのに使用する問題ファクトは、その解の実行前または実行中に変更できます。状況は複数あり、組み合わせ使用することもできます。

- 突発的なファクトの変更: シフトに入っている従業員が病気で休むと電話してきた、離陸が予定されている飛行機に技術的問題が発生して遅れている、機械または車両が壊れたなどの状況が発生した場合は、バックアップ計画 を使用します。
- 現時点ですべてのエンティティーを割り当てられない: 一部を割り当てないままにしておきます。たとえば、同時に割り当てられるシフト数が 10 個あっても、割り当てられる従業員が 9 人しかいない場合は、過制約計画 を使用します。
- 長期的未来の未知のファクト: たとえば、今後 2 週間の入院予定は確実ですが、3、4 週間後の予定になると確実ではなくなり、5 週間後以降はいま計画しても役に立ちません。継続的計画 を使用します。
- 常に変化する問題ファクト: リアルタイム計画 を使用します。

問題ファクトが変化するリスクを減らすために、計画の開始を遅らせるのは、あまり良い方法ではありません。CPU 時間を増やすことがプランニングソリューションを良くします。計画が不完全な方が、全くないよりも良くなります。

このため、最適化アルゴリズムでは、すでに (一部) 計画されている解の計画をサポートしています。これは反復計画と呼ばれています。

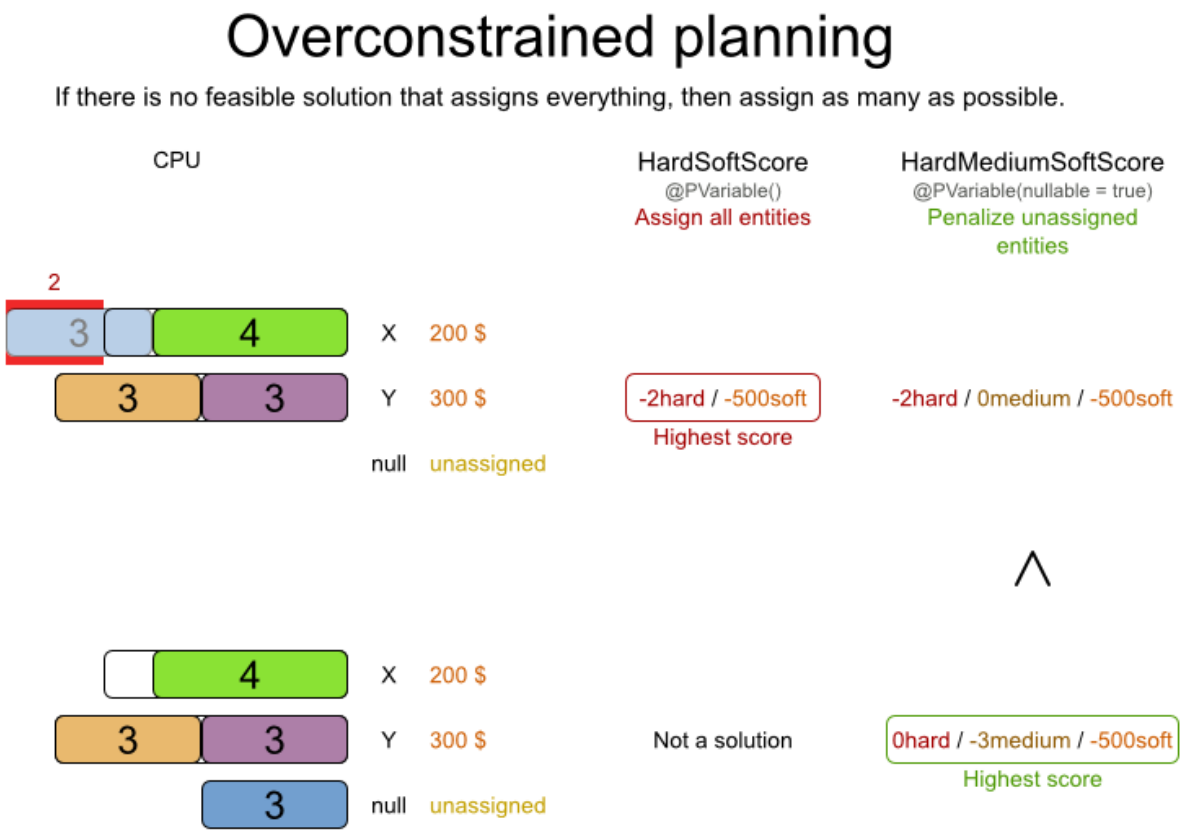
15.2. バックアップ計画

バックアップ計画は、何か問題が発生したときのために、計画に余裕を作る余分なスコア制約を追加する技術です。これにより、計画の中にバックアップ計画が作成されます。たとえば、従業員を予備の従業員として割り当て (同じ時間の 10 個のシフトに対して 1 つの予備)、病院のベッドを、各科に 1 つ空いたままにしておくなどです。

これにより、問題が発生したとき (従業員の 1 人が病気で電話してきた場合) に、元の解の問題ファクトを変更し (病気の従業員を削除し、そのシフトを割り当てないままにして)、計画を再度行い、スコアが変わったその解から始めます。構築ヒューリスティックは、新たに作成されたギャップを埋め (おそらく予備の従業員)、メタヒューリスティックがそれを今後さらに改善します。

15.3. 過剰制限計画

すべてのプランニングエンティティーを割り当てるのに適したソリューションがない場合は、ハード制約に違反せずに割り当てられるだけのエンティティーを割り当てることが必要になります。これは、過剰制限計画と呼ばれます。



これは、以下のように実装します。

1. ScoreDefinition を変えて、余分なスコアレベル (通常は、ハードレベルとソフトレベルとの間にある中間レベル) を追加します。
2. プランニング変数 nullable を作成します。
3. 新しいレベルにスコア制約 (通常は中間制約) を追加し、割り当てていないエンティティーにペナルティーを追加します (または、その合計に重みを追加します)。

15.4. 継続的計画 (ウィンドウがある計画)

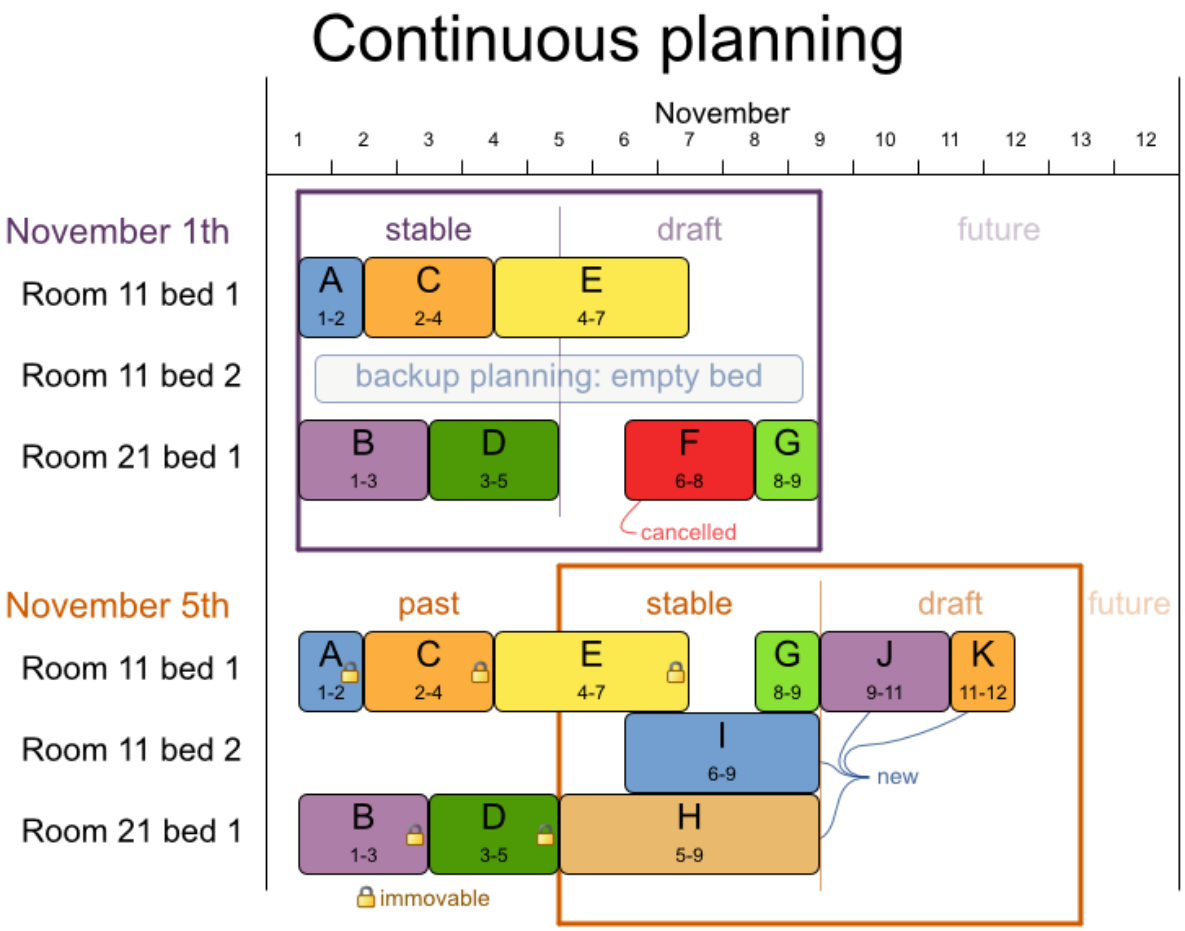
継続的計画は、同時に 1 つ以上の計画ウィンドウを、毎月、毎週、毎日、毎時に、そのプロセスを繰り返し計画するテクニックです。時間は無制限であるため、将来のウィンドウは無限になり、そのウィンドウをすべて計画するのは不可能です。したがって、一定数のウィンドウだけを計画します。

過去の計画ウィンドウは変えられません。次に行う計画ウィンドウは安定してる (変更する可能性が低い) と見なされ、その後で発生するウィンドウはドラフトと見なされます (次の計画時に変更する可能性があります)。すぐに発生しない計画ウィンドウは何も計画されていません。

過去の計画ウィンドウには、固定の 計画ウィンドウだけがあります。プランニングエンティティーは変更できなくなります (Move できないため) が、次回のプランニングエンティティーで適用するスコア制約の一部に影響する可能性があるため、その一部はスコア計算で必要になります。たとえば、従業員は、連続して 6 日以上働くことができない条件で、すでに 4 日連続して働いている場合は、本日と翌日のいずれかを休みにする必要があります。

すべてのプランニングエンティティーが常に変更できないわけではありません。変更することはできますが、元のものと異なる場合は、スコアペナルティーが発生します。たとえば、患者の入院開始日の 2

日前以降は (本当に必要な場合を除いて) 病床を再スケジュールしない、搭乗手続きの 2 時間前以降は飛行機の乗降口を変更しないなどです。



最初の計画の 11 月 1 日と、新しい計画の 11 月 5 日の違いに注目してください。問題ファクトの一部 (F、H、I、J、K) が変更したため、関係ないプランニングエンティティー (G) も変更しました。

15.4.1. 動かせないプランニングエンティティー

プランニングエンティティーを動かせないようにするには、**SelectionFilter** エンティティーを追加します。これは、エンティティーを動かせる場合は **true** を、動かせない場合は **false** を返します。

```
public class MovableShiftAssignmentSelectionFilter implements SelectionFilter<ShiftAssignment> {

    public boolean accept(ScoreDirector scoreDirector, ShiftAssignment shiftAssignment) {
        ShiftDate shiftDate = shiftAssignment.getShift().getShiftDate();
        NurseRoster nurseRoster = (NurseRoster) scoreDirector.getWorkingSolution();
        return nurseRoster.getNurseRosterInfo().isInPlanningWindow(shiftDate);
    }

}
```

設定は以下のようになります。

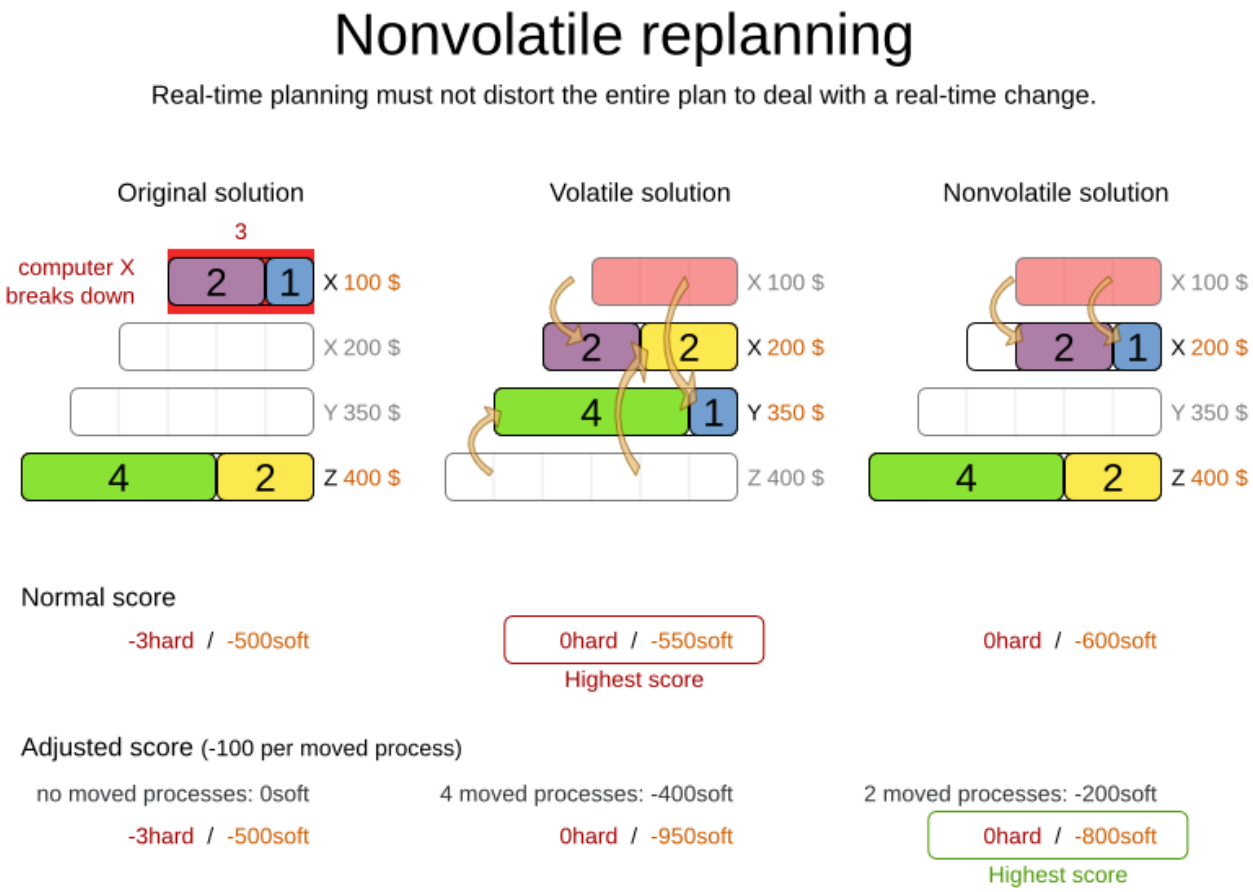
```
@PlanningEntity(movableEntitySelectionFilter = MovableShiftAssignmentSelectionFilter.class)
public class ShiftAssignment {
    ...
}
```

-

> **警告**
>
> カスタムの **MoveListFactory** および **MoveIteratorFactory** の実装では、移動できないエンティティーを移動させないようにする必要があります。

15.4.2. 中断を最小限に抑える長期的な再計画 (完全には動かせないプランニングエンティティー)

既存の計画を置き換えると、その計画が大きく中断する可能性があります。その計画が人 (従業員、ドライバーなど) に影響する場合、大抵は中断は望ましくありません。このような場合は、長期的な再計画が役に立ちます。計画を変更する利点が、それによって中断が発生するよりも大きくなります。



たとえば、マシンの再割当て例では、エンティティーにプランニング変数 **machine** と、その元の値 **originalMachine** があります。

```
@PlanningEntity(...)
public class ProcessAssignment {

    private MrProcess process;
    private Machine originalMachine;
    private Machine machine;
```

```
    public Machine getOriginalMachine() {...}

    @PlanningVariable(...)
    public Machine getMachine() {...}

    public boolean isMoved() {
        return originalMachine != null && originalMachine != machine;
    }

    ...
}
```

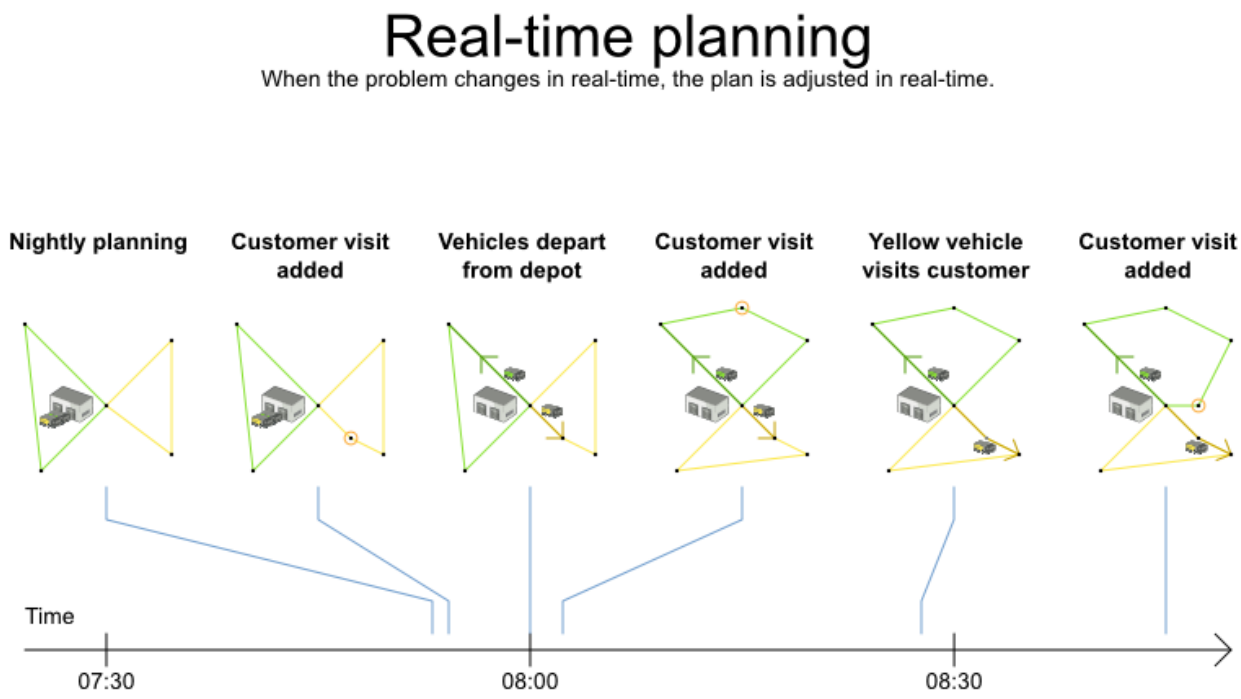
計画時、プランニング変数 **machine** が変更します。それを originalMachine と比較することで、計画における変更にペナルティーが追加されます。

```
rule "processMoved"
    when
        ProcessAssignment(moved == true)
    then
        scoreHolder.addSoftConstraintMatch(kcontext, -1000);
end
```

ソフトペナルティー **-1000** は、変更した各変数で、ソフトスコアが **1000** ポイント以上改善する場合 (またはハードスコアが改善する場合) は、最適解だけが許可されることを示しています。

15.5. リアルタイム計画

リアルタイム計画を行うには、最初に バックアップ計画 と 継続的計画 を短期の計画ウィンドウと組み合わせて、リアルタイム計画の負荷を下げます。時間が過ぎると、問題が変化します。



上の例では、元々 **07:55** に解決が終了するように顧客が設定されていたり、車が出発した後などに、顧客を 3 人、異なる時間 (**07:56**、**08:02**、および **08:45**) に追加しました。Planner は、(移動できないプランニングエンティティー とともに) **ProblemFactChange** を使用したこのようなシナリオを処理します。

15.5.1. ProblemFactChange

Solver が解決中に、外部イベントにより問題ファクトを変更する場合があります。たとえば、飛行機が遅れたために、滑走路の使用を遅らせる必要がある場合です。(別のスレッドから、もしくは同じスレッドでも) **Solver** を使用して解決中に問題ファクトのインスタンスを変更しないでください。変更すると壊れます。代わりに、**ProblemFactChange** を **Solver** に追加して、Solver スレッドをできるだけ早く実行します。

```
public interface Solver {

    ...

    boolean addProblemFactChange(ProblemFactChange problemFactChange);

    boolean isEveryProblemFactChangeProcessed();

    ...

}
```

```
public interface ProblemFactChange {
```

```
    void doChange(ScoreDirector scoreDirector);

}
```

以下は例になります。

```
    public void deleteComputer(final CloudComputer computer) {
        solver.addProblemFactChange(new ProblemFactChange() {
            public void doChange(ScoreDirector scoreDirector) {
                CloudBalance cloudBalance = (CloudBalance) scoreDirector.getWorkingSolution();
                // First remove the problem fact from all planning entities that use it
                for (CloudProcess process : cloudBalance.getProcessList()) {
                    if (ObjectUtils.equals(process.getComputer(), computer)) {
                        scoreDirector.beforeVariableChanged(process, "computer");
                        process.setComputer(null);
                        scoreDirector.afterVariableChanged(process, "computer");
                    }
                }
                // A SolutionCloner does not clone problem fact lists (such as computerList)
                // Shallow clone the computerList so only workingSolution is affected, not bestSolution or
guiSolution
                cloudBalance.setComputerList(new ArrayList<CloudComputer>
(cloudBalance.getComputerList()));
                // Next remove it the problem fact itself
                for (Iterator<CloudComputer> it = cloudBalance.getComputerList().iterator(); it.hasNext(); )
{
                    CloudComputer workingComputer = it.next();
                    if (ObjectUtils.equals(workingComputer, computer)) {
                        scoreDirector.beforeProblemFactRemoved(workingComputer);
                        it.remove(); // remove from list
                        scoreDirector.afterProblemFactRemoved(workingComputer);
                        break;
                    }
                }
            }
        });
    }
```

警告

ProblemFactChange の問題ファクトまたはプランニングエンティティーに加えた変更は、**ScoreDirector** に通知する必要があります。

重要

ProblemFactChange を正しく記述するには、計画のクローンの振る舞いを正しく理解している必要があります。

- **ProblemFactChange** における変更は、**scoreDirector.getWorkingSolution()** の **Solution** インスタンスで行う必要があります。**workingSolution** が、**BestSolutionChangedEvent** の **bestSolution** に対する 計画クローン です。したがって、**Solver** の **workingSolution** は、残りのアプリケーションの **Solution** と同じインスタンスにはなりません。
- 計画クローンは、プランニングエンティティーとプランニングエンティティーコレクションのクローンも作成します。プランニングエンティティーへの変更は、**scoreDirector.getWorkingSolution()** が保持するインスタンスで行う必要があります。
- 計画クローンは、問題ファクトのクローンでも、問題ファクトのコレクションでもありません。したがって、**workingSolution** と **bestSolution** は、同じ問題ファクトインスタンスと、同じ問題ファクトリストインスタンスを共有します。**ProblemFactChange** が変更した問題ファクトまたは問題ファクトリストは、最初にクローンを作成した問題です (これは、他の問題ファクトとプランニングエンティティーでルート変更参照を意味しています)。そうでない場合は、**workingSolution** および **bestSolution** を異なるスレッド (たとえば Solver スレッドと GUI イベントスレッド) で使用する場合に、競合状態が発生します。

注記

多くの変更では、プランニングエンティティーを初期化せず、部分的に初期化した解 (ソリューション) になります。最初の Solver フェーズで対応できれば問題ではありません。構築ヒューリスティックのすべての Solver フェーズでこれに対応できるため、最初のフェーズで、そのようなフェーズを設定することが推奨されます。

基本的に、**Solver** が停止したら、**ProblemFactChange** を実行して再起動します。これは、その初期解は 1 つ前の実行で調整した最適解になるため、ウォームスタート です。これは、構築ヒューリスティックが再度実行したことを示しますが、プランニング変数がほとんどもしくは全く初期化されないため (null 許容型のプランニング変数 がある場合を除きます)、それは、コールドスタートよりもはるかに速くなります。

(Solver およびフェーズ設定の両方に) 設定した各 終了 はリセットされ、**terminateEarly()** への以前の呼び出しは未完成ではありません。ただし、通常、(デーモンモード以外で) 終了 を設定せず、結果が必要な場合のみ **Solver.terminateEarly()** を呼び出します。代わりに、以下に示すように、終了 を設定し、**BestSolutionChangedEvent** と組み合わせてデーモンモードを使用します。

15.5.2. デーモン: solve() Does Not Return

リアルタイム計画では、作業がなくなったときに Solverスレッドが待機し、問題ファクトが新たに変更すると問題解決がすぐに再開するのが便利なことが多いです。デーモンモードに **Solver** を追加すると以下の効果があります。

- **Solver** の 終了 が終了しても、**solve()** からは返らず、そのスレッドをブロックします (これにより CPU パワーが解放されます)。
  - **solve()** からそれを返す **terminateEarly()** を除いて、システムリソースを解放し、アプリケーションが適切にシャットダウンできるようになります。

    - **Solver** が、空のプランニングエンティティーコレクションから開始する場合は、すぐにブロック状態で待機します。

- **ProblemFactChange** が追加されると、実行状態になり、**ProblemFactChange** が適用され、**Solver** を再度実行します。

デーモンモードを設定するには、以下のように設定します。

```
<solver>
  <daemon>true</daemon>
  ...
</solver>
```

警告

Solver スレッドが不自然に強制終了するのを避けるためにアプリケーションをシャットダウンする必要がある場合は、**Solver.terminateEarly()** を呼び出す必要があります。

BestSolutionChangedEvent をサブスクライブして、Solver スレッドが見つけた新しい最適解を処理します。**BestSolutionChangedEvent** は、すべての **ProblemFactChange** がすでに処理されていることを保証できず、その解が初期化され実行できる状態でもありません。そのような無効な解を持つ **BestSolutionChangedEvents** を無視するには、以下を行います。

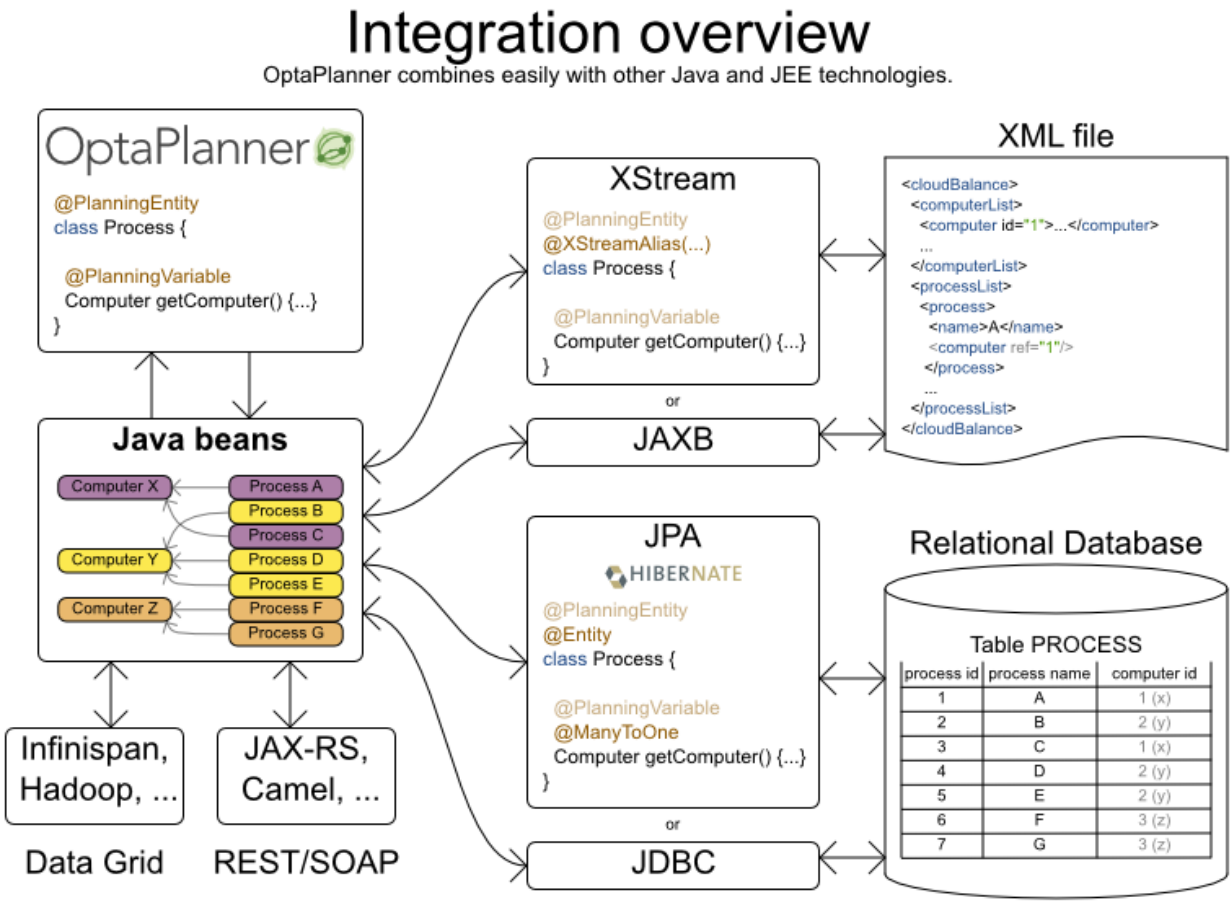
```
public void bestSolutionChanged(BestSolutionChangedEvent<CloudBalance> event) {
    // Ignore invalid solutions
    if (event.isEveryProblemFactChangeProcessed()
            && event.isNewBestSolutionInitialized()
            && event.getNewBestSolution().getScore().isFeasible()) {
        ...
    }
}
```

第16章 統合

16.1. 概要

Planner が入力および出力したデータ (計画問題と最適解) は POJO (plain old JavaBeans) であるため、その他の Java テクノロジーとの統合は単純です。たとえば以下のようになります。

- 計画問題をデータベースから読み込み、そこに最適解を保存するには、JPA アノテーションを持つドメイン POJO をアノテートします。
- XML ファイルから計画問題を読み込み、そこに最適解を保存するには、XStream アノテーションまたは JAXB アノテーションを持つドメインの POJO をアノテートします。
- Solver を、計画問題を読み込み、最適解で応答する REST サービスとして公開するには、XStream アノテーションまたは JAXB アノテーションを持つドメイン POJO をアノテートし、Camel または RESTEasy で **Solver** をフックします。



16.2. 永続ストレージ

16.2.1. データベース: JPA および Hibernate

データベースに保存するために、ドメイン POJO (ソリューション、エンティティー、問題ファクト) を JPA アノテーションで改良します。

注記

JPA の **@Entity** アノテーションと、Planner の **@PlanningEntity** アノテーションを混在しないでください。両方とも、同じクラスで使用できます。

```
@PlanningEntity // OptaPlanner annotation
@Entity // JPA annotation
public class Talk {...}
```

追加の統合機能を利用するために、**optaplanner-persistence-jpa** JAR に依存関係を追加します。

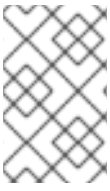
16.2.1.1. JPA および Hibernate: スコアの永続化

スコア がリレーショナルデータベースに永続化される場合は、JPA および Hibernate がデフォルトで **BLOB** カラムにシリアル化します。これには、いくつかのデメリットがあります。

- **Score** クラスの Java のシリアル化形式は現在、後方互換性がありません。新しい Planner バージョンにアップグレードすると、既存のデータベースを読み込む際に破損する可能性があります。
- データベースのコンソールで実行されたクエリーでは、スコアを簡単に読み取ることができません。これでは、開発時に不便です。
- SQL または JPA-QL クエリーでスコアを使用して、その結果にフィルターを設定することはできません。たとえば、実行できないスケジュールをすべてクエリーするには、以下のようにします。

この問題を回避するには、使用している スコア タイプ (**HardSoftScore** など) に適切な ***ScoreHibernateType** を使用して、2 つの **INTEGER** 列を使用するように設定します。

```
@PlanningSolution
@Entity
@TypeDef(defaultForType = HardSoftScore.class, typeClass = HardSoftScoreHibernateType.class)
public class CloudBalance implements Solution<HardSoftScore> {

    @Columns(columns = {@Column(name = "hardScore"), @Column(name = "softScore")})
    protected HardSoftScore score;

    ...
}
```



注記

スコアに、スコアレベルの数と同じだけ **@Column** アノテーション数を設定します。設定しないと、プロパティーマッピングで列数が正しくなくなるため、Hibernate でフェイルファーストが発生します。

この場合、DDL は以下のようになります。

```
CREATE TABLE CloudBalance(
    ...
    hardScore INTEGER,
```

```
    softScore INTEGER
);
```

BigDecimal ベースの **Score** を使用する場合は、列の精度とスケールを指定して、通知なく丸めが行われることを回避します。

```
@PlanningSolution
@Entity
@TypeDef(defaultForType = HardSoftBigDecimalScore.class, typeClass =
HardSoftBigDecimalScoreHibernateType.class)
public class CloudBalance implements Solution<HardSoftBigDecimalScore> {

    @Columns(columns = {
            @Column(name = "hardScore", precision = 10, scale = 5),
            @Column(name = "softScore", precision = 10, scale = 5)})
    protected HardSoftBigDecimalScore score;

    ...
}
```

丸められる スコア のタイプを使用して、ハードおよびソフトのレベルサイズをパラメーターとして指定します。

```
@PlanningSolution
@Entity
@TypeDef(defaultForType = BendableScore.class, typeClass = BendableScoreHibernateType.class,
parameters = {
        @Parameter(name = "hardLevelsSize", value = "3"),
        @Parameter(name = "softLevelsSize", value = "2")})
public class Schedule implements Solution<BendableScore> {

    @Columns(columns = {
            @Column(name = "hard0Score"),
            @Column(name = "hard1Score"),
            @Column(name = "hard2Score"),
            @Column(name = "soft0Score"),
            @Column(name = "soft1Score")})
    protected BendableScore score;

    ...
}
```

このサポートは Hibernate に固有のものです。複数列に変換するのをサポートしていないため、現時点では、JPA 2.1 のコンバーターではサポートしていません。

16.2.1.2. JPA および Hibernate: 計画のクローン作成

JPA および Hibernate では、多くの問題ファクトクラスからプランニングソリューションクラスへの、**@ManyToOne** の関係があります。したがって、問題ファクトクラスは、プランニングソリューションクラスを参照します。したがって、そのソリューションが クローンが作成された計画 の場合は、問題ファクトクラスのクローンも作成する必要があります。そのような各問題ファクトクラスで **@DeepPlanningClone** を使用して、それを強化します。

```
@PlanningSolution // OptaPlanner annotation
@Entity // JPA annotation
```

```
public class Conference {

    @OneToMany(mappedBy = "conference")
    private List<Room> roomList;

    ...
}
```

```
@DeepPlanningClone // OptaPlanner annotation: Force the default planning cloner to planning clone this class too
@Entity // JPA annotation
public class Room {

    @ManyToOne
    private Conference conference; // Because of this reference, this problem fact needs to be planning cloned too

}
```

これを行わないと、ソリューションが重複したり、JPA 例外が発生したりなどの問題が発生する可能性があります。

16.2.2. XML または JSON: XStream

ドメイン POJO (ソリューション、エンティティー、および問題ファクト) を XStream アノテーションで改良して、XML または JSON とのやりとりでシリアル化します。

追加の統合機能を利用するために、依存関係を **optaplanner-persistence-xstream** JAR に追加します。

16.2.2.1. XStream: スコアのマーシャリング

デフォルトの XStream 設定で、スコア が XML または JSON にマーシャリングされると、非常に長くなり扱いにくくなります。これを修正するには、**XStreamScoreConverter** で、 **ScoreDefinition** をパラメーターとして設定します。

```
@PlanningSolution
@XStreamAlias("CloudBalance")
public class CloudBalance implements Solution<HardSoftScore> {

    @XStreamConverter(value = XStreamScoreConverter.class, types = {HardSoftScoreDefinition.class})
    private HardSoftScore score;

    ...
}
```

たとえば、以下のように設定すると、良質の XML が生成されます。

```
<CloudBalance>
  ...
  <score>0hard/-200soft</score>
</CloudBalance>
```

丸められるスコアのタイプにこれを使用するには、**hardLevelsSize** および **softLevelsSize** を定義する 2 つの **int** パラメーターを提供します。

```
@PlanningSolution
@XStreamAlias("Schedule")
public class Schedule implements Solution<BendableScore> {

    @XStreamConverter(value = XStreamScoreConverter.class, types =
{BendableScoreDefinition.class}, ints = {2, 3})
    private BendableScore score;

    ...
}
```

たとえば、以下を生成します。

```
<Schedule>
  ...
  <score>0/0/-100/-20/-3</score>
</Schedule>
```

16.2.3. XML または JSON: JAXB

JAXB アノテーションを使用して、ドメイン POJO (ソリューション、エンティティー、問題ファクト) を改良して、XML または JSON とシリアル化します。

16.3. SOA および ESB

16.3.1. Camel および Karaf

Camel (Camel 2.13 以降) は、Planner のサポートが含まれるエンタープライズインテグレーションフレームワークです。これは、ユースケースを REST サービス、SOAP サービス、JMS サービスなどとして公開します。

camel-optaplanner コンポーネントのドキュメントを参照してください。Karaf でもそのコンポーネントは機能します。

16.4. その他の環境

16.4.1. JBoss モジュール、WildFly、および JBoss EAP

WildFly に Planner web アプリケーションをデプロイする場合は、**optaplanner-webexamples-*.war** に記載しているように、(その他の依存関係と同じように) WAR ファイルの **WEB-INF/lib** ディレクトリーに optaplanner 依存関係 JAR を追加します。ただし、この方法では、WAR ファイルのサイズが簡単に数 MB にまで増えます。1 回のデプロイメントではこのサイズでも問題はありませんが、(特にネットワーク接続が遅い場合は) が重量すぎるためデプロイメントを何度もやりなおすことはできません。

この問題を解決するには、optaplanner JAR を JBoss module として WildFly に配置し、軽量な WAR を作成します。**org.optaplanner** という名前のモジュールを作成します。

1. **${WILDFLY_HOME}/modules/system/layers/base/** ディレクトリーに移動します。このディレクトリーには、WildFly の JBoss モジュールが含まれます。新しいモジュール用に **org/optaplanner/main** ディレクトリー構造を作成します。

a. **optaplanner-core-${version}.jar** と、その直接および推移的な依存関係の JAR を、新しいディレクトリーにコピーします。各 optaplanner アーティファクトで **mvn dependency:tree** コマンドを使用して、すべての依存関係を検出します。

b. 新しいディレクトリーに **module.xml** を作成し、そのファイルに以下を記載します。

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="org.optaplanner">
  <resources>
    ...
    <resource-root path="kie-api-${version}.jar"/>
    ...
    <resource-root path="optaplanner-core-${version}.jar"/>
    ...
    <resource-root path="."/>
  </resources>
  <dependencies>
    <module name="javaee.api"/>
  </dependencies>
</module>
```

2. デプロイした WAR ファイルに移動します。

a. **optaplanner-core-${version}.jar** と、その直接および推移的な依存関係の JAR を、WAR ファイルの **WEB-INF/lib** ディレクトリーから削除します。

b. **WEB-INF/lib** ディレクトリーに **jboss-deployment-structure.xml** ファイルを作成します。そのファイルに以下を記載します。

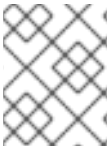
```
<?xml version="1.0" encoding="UTF-8" ?>
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.optaplanner" export="true"/>
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

JBoss モジュールの **ClassLoader** magic があるため、SolverFactory 作成時 に、お使いのクラスの **ClassLoader** を提供することがおそらく必要になります。これにより、(Solver 設定、スコア DRL とドメインのクラスなど) JAR の classpath リソースを見つけることができます。

16.4.2. OSGi

OSGi 環境でも適切に機能するように、**optaplanner-core** JAR は、その **MANIFEST.MF** に OSGi メタデータを追加します。さらに、Maven アーティファクト **drools-karaf-features** (名前が **kie-karaf-features** に変更になります) には、OSGi 機能 **optaplanner-engine** をサポートする **features.xml** ファイルが含まれます。

OSGi の **ClassLoader** magic があるため、SolverFactory 作成時 に、お使いのクラスの **ClassLoader** を提供することがおそらく必要になります。これにより、(Solver 設定、スコア DRL とドメインのクラスなど) JAR の classpath リソースを見つけることができます。



注記

Planner には OSGi が必要ありません。これは、通常の Java 環境でも問題なく動作します。

16.4.3. Android

(一部の JDK ライブラリーが不足しているため) Android は完全な JVM ではありませんが、Planner は Android でも、easy Java または Java インクリメント演算子 によるスコア計算を使用すれば機能します。Android では Drools ルールエンジンに対応しておらず、 Drools スコア計算は Android では機能しないため、その依存関係は削除する必要があります。

Android で Planner を使用するための回避策:

1. 依存関係を、使用する Android プロジェクトの **build.gradle** ファイルに追加して、**org.drools** と **xmlpull** の依存関係を除外します。

```
dependencies {
    ...
    compile('org.optaplanner:optaplanner-core:...') {
        exclude group: 'xmlpull'
        exclude group: 'org.drools'
    }
    ...
}
```

16.5. PLANNER と手動での計画の統合 (駆け引き)

Planner の実装が適切に行われていて、単純ではないデータセットを使用した場合は、手動での計画よりも Planner の方が良い結果が得られます。専門家の多くがこれを認めませんが、それは、自動化システムを脅威だと思っているからです。

実際には、専門家が Planner の監督者となると、両方に対する利益になります。

- 手動で、スコア関数を定義および検証します。
    - 簡単なスコア制約に対する重みを定義する **Parametrization** オブジェクトが公開されている例もあります。実行時に手動でこの重みを調整できます。
    - ビジネスが変更したら、多くの場合、スコア関数も変更する必要があります。担当者は、スコア制約を追加、変更、または削除するのを開発者に知らせることができます。
- 人間が、常に Planner を管理します。
    - 「コースの時間割」の例で示しているように、手動で、1 つ以上のプランニング変数を、特定の計画値にロックして動かせないようにすることができます。この変数は 動かせない ため、Planner は変更できません。手動で強制的に行われた設定により、計画が最適化されます。手動ですべてのプランニングエンティティーをロックしたら、Planner を使用する必要はなくなります。
    - プロトタイプの実装では、この方法が使用されることもありますが、実装が十分に成長したら、この方法は選択されなくなります。ただ、手動で導いた結果を再確認するために、この機能は動かしたままにします。または、スコア制約が調整される前にビジネスが大幅に変更するかもしれません。

したがって、実際のプロジェクトでは、多くの場合、Planner と人間を共存することが推奨されます。

第17章 計画パターン

17.1. 計画パターンの導入

ここで挙げる計画パターンは、計画時に発生する一般的な課題をリストアップして解決します。

17.2. 計画エンティティへの時間の割り当て

時間と日程は、ユースケースのニーズによって異なるため、計画問題で時間と日数を取り組むのは簡単ではありません。

Java には、タイムスタンプ、日数、期間を表現する方法はいくつかあります。ユースケースに適した表現を選択します。

- **java.util.Date** (deprecated): タイムスタンプの表示方法。時間がかかり、間違いが起こりやすいため、使用しないでください。
- **javax.time.LocalDateTime**、**LocalDate**、**DayOfWeek**、**Duration**、**Period** など: タイムスタンプや日数などを正確に表示して計算する方法。
  - タイムゾーンと夏時間 (DST) に対応します。
  - Java 8 以降が必要です。
    - Java 7 の場合は、**ThreeTen Backport** という名前のバックポートを使用します。
    - Java 6 以前の場合は、代わりに前身となる **Joda-Time** を使用します。
- **int** または **long**: 粒度の粗い時間単位 (分など) の簡易数値として、グローバル計画の時間枠またはエポックの開始時からのタイムスタンプをキャッシュに格納します。
  - たとえば、**LocalDateTime** の **1-JAN 08:00:00** と **1-JAN 09:00:00** は、**int** ではそれぞれ **400** 分と **460** 分になります。
  - これは、しばしばクラスの別フィールドと、それから計算した **LocalDateTime** フィールドを表示します。ユーザーへの可視化には **LocalDateTime** が使用されますが、スコア制約には **int** が使用されます。
  - 計算が速く、特に TimeGrain pattern で便利です。
  - タイムゾーンまたは DST がスコア制約に影響する場合は使用しないでください。

また、計画エンティティを開始時間 (または開始日) に割り当てる計画がいくつかあります。

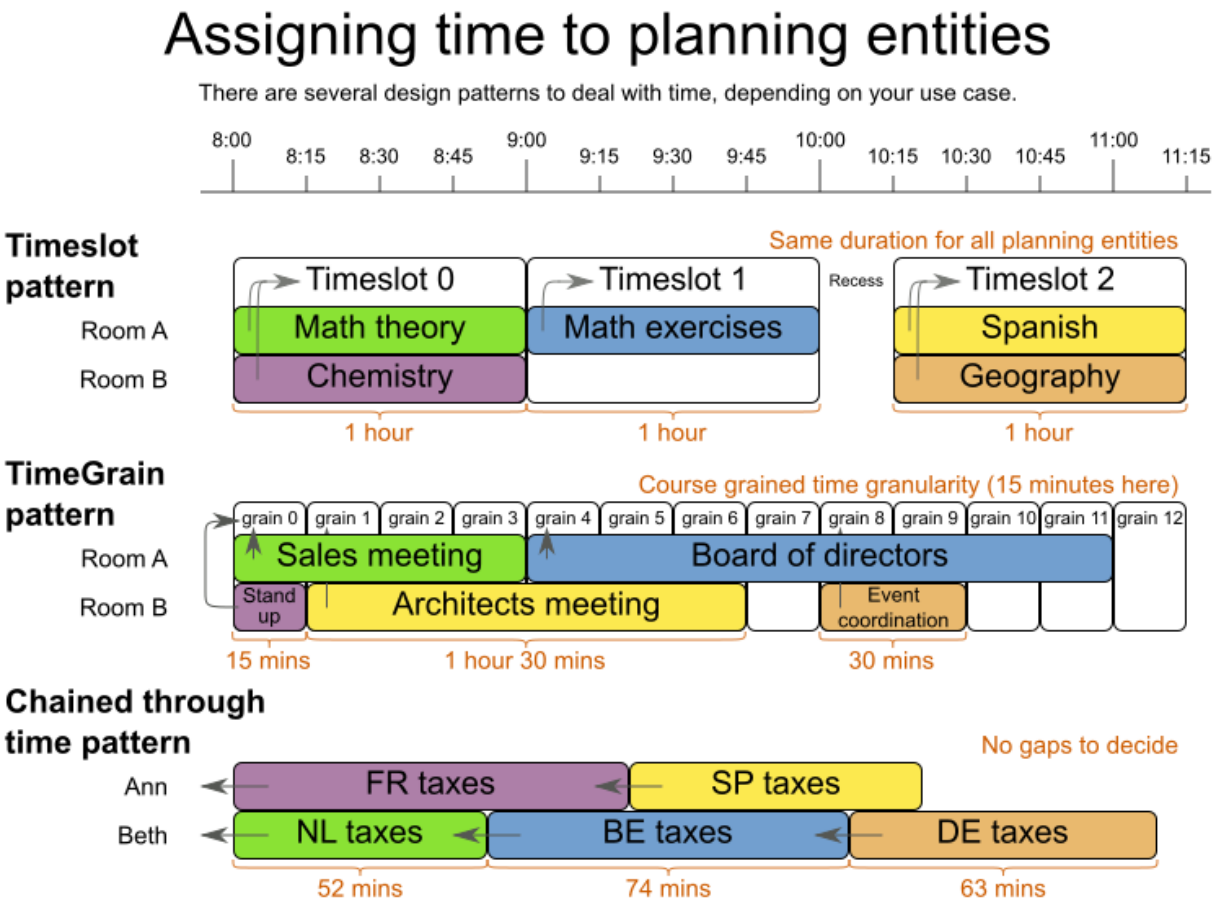
- 開始時間が前もって決められています。(そのような Solverでは) これはプランニング変数ではありません。
  - たとえば、病床計画 例では、患者が入院する日は前もって決まっています。
  - これは、多段階計画ではよくあることで、計画の前段階ですでに開始時間が決まっています。
- 開始時間が固定されていない場合は、プランニング変数です (正規またはシャドウ)。
  - すべての計画エンティティの期間が同じ場合は、Timeslot パターン を使用します。

- たとえば、コースの時間割では、講義の長さはすべて 1 時間なので、時間割の長さは 1 時間になります。

- 長さが異なり、特定の間隔 (たとえば 5 分間隔) で区切る場合は、TimeGrain パターンを使用します。
  - たとえば、会議のスケジューリングでは、すべての会議が 15 分間隔で開始し、長さは 15 分、30 分、45 分、60 分、90 分、または 120 分になります。

- 長さが異なり、(同じ実行者に割り当てられた) 前のタスクが終了したらすぐに別のタスクが開始する場合は、Chained Through Time パターンを使用します。
  - たとえば、時間枠での車両の決定では、前の顧客への配送が終了したらすぐに次の顧客に向かいます。

このユースケースに適切なパターンを選択します。



17.2.1. Timeslot パターン: 固定長の時間枠に割り当て

プランニングエンティティーの期間がすべて同じ (または、同じ期間に拡張できる) 場合は、時間枠パターンが便利です。プランニングエンティティーを、時間ではなく時間枠に割り当てます。たとえば、コースの時間割では、すべての授業が 1 時間になります。

時間枠はいつでも開始できます。たとえば、時間枠を 8:00、9:00、(15 分の休憩)、10:15、11:15 に開始します。重ねることはできますが、通常は重ねません。

また、すべてのプランニングエンティティーを同じ期間に拡張することができます。たとえば 試験の時間割では、90 分のものもあれば、120 分のものもありますが、時間枠はすべて 120 分です。90 分の試

験を時間枠に割り当てる場合、残りの 30 分は席が予約されたままになり、別の試験で使用することはできません。

通常、2 つ目のプランニング変数 (たとえば部屋) があります。コースの時間割では、2 つの授業が同じ部屋を同じ時間枠で共有すると競合しますが、試験の時間割では、その部屋に席が余っていれば許可されます (ただし、同じ部屋で複数の試験を行う期間では、ソフトスコアペナルティーが課せられます)。

17.2.2. TimeGrain パターン: 開始 TimeGrain に割り当て

9 時 4 秒に会議を開始するように人を割り当てるのは役に立ちません。なぜなら、人の活動における時間粒度は 5 分または 15 分だからです。したがって、プランニングエンティティーを、1 秒未満、1 秒、または 1 分の精度で割り当てる必要はありません。5 分または 15 分の精度で十分です。TimeGrain パターンは、時間を時間粒として分割することでこのような時間精度を作ります。たとえば、会議のスケジュール では、すべての会議は 1 時間、30 分、または各時 15 分前後の間隔で始まるため、時間粒度の最適な設定は 15 分になります。

各プランニングエンティティーは、開始時間粒に割り当てられます。終了時間粒は、開始時間粒に追加して計算します。2 つのエンティティーが重複するかどうかは、開始時間粒と終了時間粒を比較することで決まります。

このパターンは、時間粒度が粗くても (1 日、半日、数時間など) 機能します。時間粒度が細かく (秒、ミリ秒など)、時間枠が長い場合、値の範囲 (および 探索空間) は高くなりすぎて、効率とスケーラビリティーが低くなります。ただし、コストを抑えるスケジュール で示すように、このような解は不可能ではありません。

17.2.3. Chained Through Time パターン: 開始時間を決める連鎖に割り当て

人間またはマシンが、順番に、一度に 1 つのタスクを継続して取り組む場合 (以前のタスクが終了したらタスクが開始したり、決定論的遅延がある場合)、Chained Through Time パターンは役に立ちます。たとえば、時間枠がある配車例では、車が顧客から顧客に移動します (したがって一度に 1 人の顧客を処理します)。

このパターンでは、プランニングエンティティーは 連鎖 となります。アンカーは、最初のプランニングエンティティーの開始時を決めます。2 番目のエンティティーの開始時間は、最初のエンティティーの開始時間と期間に基づいて計算されます。たとえば、タスク割り当てで、Beth (アンカー) は 8:00 に働き始め、1 つ目の仕事を 8:00 に開始します。これは 52 分続くため、2 番目の仕事は 8:52 に開始します。エンティティーの開始時間は通常 シャドウ変数 です。

アンカーには連鎖が 1 つだけあります。アンカーは 2 つに分割することができます (たとえば、Beth は同時に 2 つのタスクを行うことができるため、Beth を Beth の左手と、Beth の右手に分けます) が、このモデルでは予備のリソースを持つことが難しくなります。したがって、試験の時間割例でこのモデルを使用して、同じ時間に同じ部屋を使うように 2 つ以上の試験を割り当てることは問題となります。

プランニングエンティティーでは、ギャップを作成する方法は 3 つあります。

- ギャップなし: これは、アンカーがマシンの場合に一般的です。たとえば、ビルドサーバーは、1 つのジョブを終了したら、常に中断せずに次のジョブを開始します。
- 決定論的ギャップ: これは、人間の場合に一般的です。たとえば、10:00 の境界を超えるすべてのタスクでは、15 分間余分に割り当てることで、休憩が取れます。
    - 決定論的ギャップは、複雑なビジネスロジックに従う場合があります。たとえば、配車例では、大陸を横断するトラック運転手は、運転を 2 時間続けたら 15 分の休憩が必要になります (これは、顧客の場所で荷物の積み下ろしている間も適用されます)。また、14 時間働いたら 10 時間の休むことが必要になります。

- プランニング変数のギャップ: これは一般的ではありません。(探索空間 に影響する) プランニング変数が過剰に追加されることで、効率性とスケーラビリティーが減るためです。

17.3. 多段階計画

(コンウェイの法則 などの) 実用的または組織的な理由により、複合的な計画問題を複数の段階に分類する場合がしばしばあります。典型的な例として、電車のスケジューリングが挙げられます。ある部署が、電車の発着場所と時間を決め、別の部署が、列車の車両や機関車にどの運転手を割り当てるかを決定します。

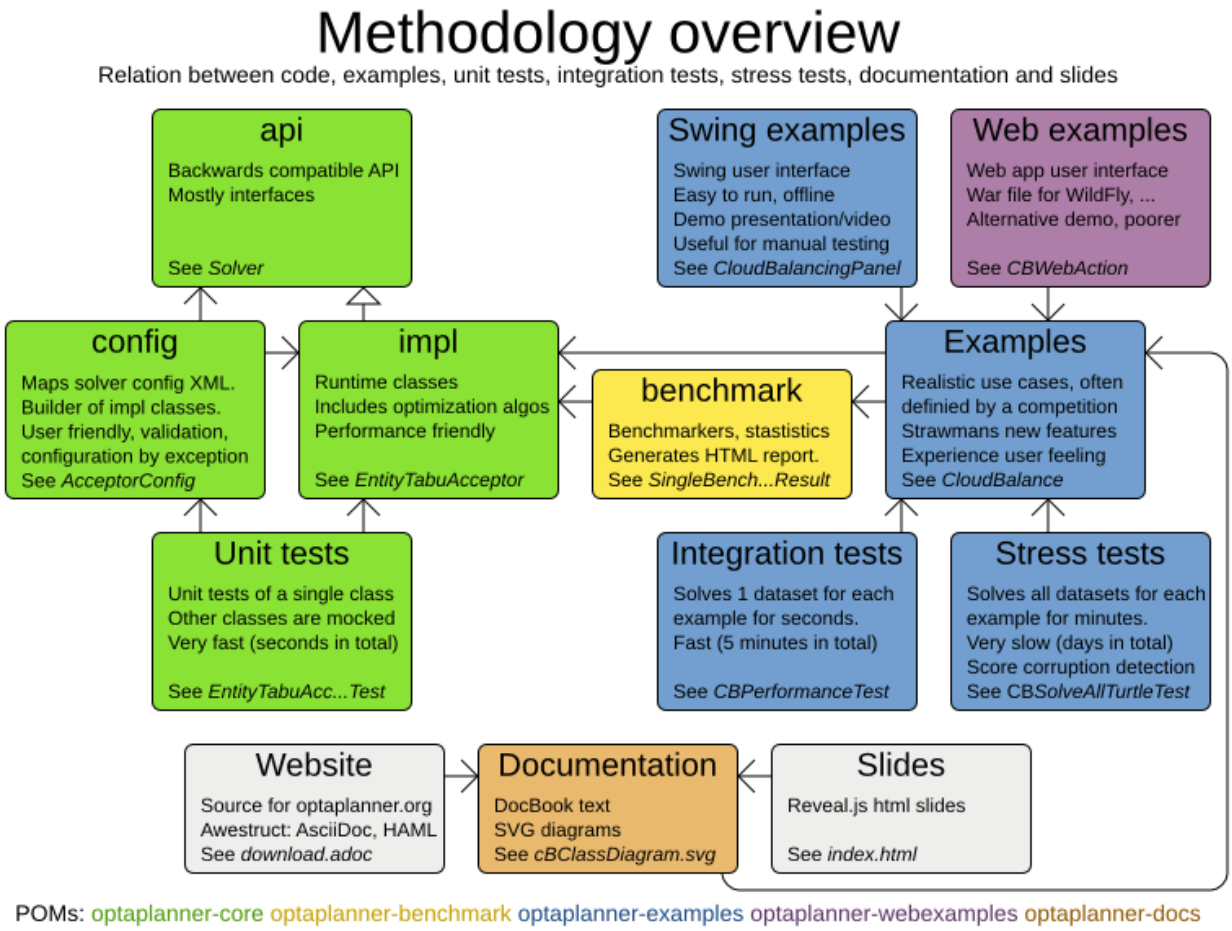
各段階に特有の Solver 設定 (つまり、特有の **SolverFactory**) があります。Solver 設定を 1 つだけ使用する 多相解決 とは混合しないでください。

分割検索 と同様、多段階計画は準最適な結果となります。それでもなお、メンテナンスや所有者を簡略化し、プロジェクトの立ち上げに貢献するのに役に立つ場合があります。

第18章 開発

18.1. 方法論の概要

以下の図は、OptaPlanner ソースコードの全体的な構造を示します。



上の図では、設定とランタイムクラスの明確な境界を理解することが重要です。

開発体系には、以下が含まれます。

- 再利用: 例は、統合テスト、ストレステスト、およびデモで再利用されます。ドキュメントのイメージはスライドで再利用されます。
- 一貫した用語: 各例には **App** クラス (実行可能なクラス)、**Dao** (Data Access Object)、 **Panel** (swing UI) が使われています。
- 一貫性した構造: 各例では同じパッケージが使われています (**domain**、**persistence**、**app**、**solver**、および **swingui**)。
- 現実世界の有用性: 例ではすべての機能が使用されています。ほとんどの例は、実在する制約を使用した実在するユースケースで、多くの場合、実在するデータが使用されています。
- 自動テスト: ユニットテスト、統合テスト、パフォーマンス不具合テスト、およびストレステストがあり、テスト対象は広範囲です。
- フェイルファーストと理解可能なエラーメッセージ: 無効な状態は、できるだけ早く確認されます。

18.2. 開発ガイドライン

1. フェイルファーストには段階がいくつかあります (重大度は、下になるほど悪くなります)。
   a. コンパイル時の失敗。たとえば、**String** または **Integer** が必要な場合に、**Object** をパラメーターとして取りません。
   b. 起動時の失敗。たとえば、設定パラメーターに正の **int** が必要な場合に負の値を設定すると失敗します。
   c. ランタイム時の失敗。たとえば、リクエストで **0.0** から **1.0** までの二重階層が必要な場合に、**1.0** より大きい値を設定すると失敗します。
   d. アサーションモードにおけるランタイム時の失敗 (検出のパフォーマンスコストが高い場合)。たとえば、低水準の反復後に必ず変数 A を B の 平方根にする必要がある場合は、アサートフラグが true に設定されている場合に限りそれを確認します (通常は EnvironmentMode で制御)。
2. 例外メッセージ
   a. 例外メッセージには、以下のように、関連する変数の名前およびステータスが含まれます。

```
if (fooSize < 0) {
    throw new IllegalArgumentException("The fooSize (" + fooSize + ") of bar (" + this + ")
must be positive.");
}
```

   出力では、問題が明確に説明されています。

```
Exception in thread "main" java.lang.IllegalArgumentException: The fooSize (-5) of bar
(myBar) must be positive.
    at ...
```

   b. 可能な限り、例外メッセージにはその背景が含まれます。
   c. 修正方法が明らかではない場合は、例外メッセージにアドバイスが含まれます。アドバイスは通常、行が変わり、 **maybe** という単語から始まります。

```
Exception in thread "main" java.lang.IllegalStateException: The valueRangeDescriptor
(fooRange) is nullable, but not countable (false).
Maybe the member (getFooRange) should return CountableValueRange.
    at ...
```

   maybe という単語は、そのアドバイスがすべてのケースで正しいことが保証できないことを示しています。
3. ジェネリック型。しばしば、**Solution** クラスがジェネリック型パラメーターとしてサブシステムに渡されます。**PlanningEntity** クラスがジェネリック型パラメーターとして渡されることはほとんどありません。

第19章 移行ガイド

19.1. BUSINESS RESOURCE PLANNER の移行の概要

Business Resource Planner は、Red Hat JBoss BRMS および Red Hat JBoss BPM Suite の 6.1 以降で完全に対応しています。Business Resource Planner は、計画問題を解決できる、軽量で、埋め込み可能な制約充足エンジンです。Business Resource Planner にはパブリック API が含まれ、6.2 や 6 などの後続バージョンに後方互換が組み込まれます。詳細は、Business Resource Planner ドキュメンテーションを参照してください。

JBoss BPM Suite および JBoss BRMS の 6.1 以降に、以下の変更が実装されました。

- 焼きなまし法では、最後のステップではなく、現在のステップの時間勾配が使用されます。この変更の影響はごくわずかです。
- AbstractScore では、**parseLevelStrings(...)** メソッドおよび **buildScorePattern(...)** メソッドが、public から protected に変更になりました。コードがこの変更の影響を受ける可能性は極めて低いです。
- **Descriptor** クラスのパッケージが、記述子パッケージへ変更になりました。**SolutionDescriptor.isInitialized(Solution)** には **ScoreDirector** パラメーターが必要になります。
- 力まかせ (Brute Force) よりも優れた選択肢として分岐限定 (Branch And Bound) があります。詳細はドキュメンテーションを参照してください。
- **InverseRelationShadowVariableListener** の名前が **SingletonInverseVariableListener** に変更になり、**InverseRelationShadowVariableDescriptor** とともに、パッケージが **...impl.domain.variable.inverserelation** に変更になりました。
- (非推奨になっていた) **ConstraintOccurrence** クラスが削除されたため、**ConstraintMatch** システムに切り替える必要があります。
- **Solution** インターフェースがパブリック API になり、パッケージが **impl.solution** から **api.domain.solution** へ変更になりました。
  以前の ***.java** :

  ```
  import org.optaplanner.core.impl.solution.Solution;
  ```

  現在の ***.java** :

  ```
  import org.optaplanner.core.api.domain.solution.Solution;
  ```

- **impl.move** パッケージのクラスはすべて **impl.heuristic.move** に変更になりました。現時点では、パブリック API に追加されるかどうかは予測できないため、できる限り一般的な Move を使用することが推奨されます。
  以前の ***.java** :

  ```
  import org.optaplanner.core.impl.move.Move;
  import org.optaplanner.core.impl.move.CompositeMove;
  import org.optaplanner.core.impl.move.NoChangeMove;
  ```

  現在の ***.java** :

```
import org.optaplanner.core.impl.heuristic.move.Move;
import org.optaplanner.core.impl.heuristic.move.CompositeMove;
import org.optaplanner.core.impl.heuristic.move.NoChangeMove;
```

- すべてのクラスパスリソースにある先頭のスラッシュは削除する必要があります。Business Resource Planner では、このリソースは、**Class.getResource(String)** ではなく **ClassLoader.getResource(String)** を遵守する必要があります。
  - **SolverFactory.createFromXmlResource(String)** パラメーターの、先頭のスラッシュは削除する必要があります。
    以前の ***.java** :

    ```
    ... = SolverFactory.createFromXmlResource(
    "/org/optaplanner/examples/cloudbalancing/solver/cloudBalancingSolverConfig.xml");
    ```

    現在の ***.java** :

    ```
    ... = SolverFactory.createFromXmlResource(
    "org/optaplanner/examples/cloudbalancing/solver/cloudBalancingSolverConfig.xml");
    ```

  - **<scoreDrl>** のすべての要素で、先頭のスラッシュは削除する必要があります。
    以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

    ```
    <scoreDrl>/org/optaplanner/examples/cloudbalancing/solver/cloudBalancingScoreRules.drl
    </scoreDrl>
    ```

    現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

    ```
    <scoreDrl>org/optaplanner/examples/cloudbalancing/solver/cloudBalancingScoreRules.drl
    </scoreDrl>
    ```

  - **PlannerBenchmarkFactory.createFromXmlResource(String)** パラメーターの、先頭のスラッシュは削除する必要があります。
    以前の ***.java** :

    ```
    ... = PlannerBenchmarkFactory.createFromXmlResource(

    "/org/optaplanner/examples/cloudbalancing/benchmark/cloudBalancingBenchmarkConfig.x
    ml");
    ```

    現在の ***.java** :

    ```
    ... = PlannerBenchmarkFactory.createFromXmlResource(

    "org/optaplanner/examples/cloudbalancing/benchmark/cloudBalancingBenchmarkConfig.xm
    l");
    ```

  - **PlannerBenchmarkFactory.createFromFreemarkerXmlResource(String)** パラメーターの、先頭のスラッシュは削除する必要があります。
    以前の ***.java** :

```
... = PlannerBenchmarkFactory.createFromFreemarkerXmlResource(
"/org/optaplanner/examples/cloudbalancing/benchmark/cloudBalancingBenchmarkConfigTe
mplate.xml.ftl");
```

現在の ***.java** :

```
... = PlannerBenchmarkFactory.createFromFreemarkerXmlResource(
"org/optaplanner/examples/cloudbalancing/benchmark/cloudBalancingBenchmarkConfigTe
mplate.xml.ftl");
```

- 連鎖された **@PlanningVariable** プロパティは、**graphType** にリファクタリングされました。これは、将来的に別のグラフタイプ (TREE など) に対応できるようにするためです。
以前の ***.java** :

```
@PlanningVariable(chained = true, ...)
public Standstill getPreviousStandstill() {
    return previousStandstill;
}
```

現在の ***.java** :

```
@PlanningVariable(graphType = PlanningVariableGraphType.CHAINED, ...)
public Standstill getPreviousStandstill() {
    return previousStandstill;
}
```

- **constructionHeuristicType** の **BEST_FIT** の名前が **WEAKEST_FIT** に変更になりました。「Best Fit (最良適合)」という用語は適切ではなく、**STRONGEST_FIT** は許可されませんでした。
以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<constructionHeuristic>
  <constructionHeuristicType>BEST_FIT</constructionHeuristicType>
</constructionHeuristic>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<constructionHeuristic>
  <constructionHeuristicType>WEAKEST_FIT</constructionHeuristicType>
</constructionHeuristic>
```

- **constructionHeuristicType** の **BEST_FIT_DECREASING** の名前が **WEAKEST_FIT_DECREASING** になりました。「Best Fit (最良適合)」という用語は適切ではなく、**STRONGEST_FIT_DECREASING** は許可されませんでした。
以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<constructionHeuristic>
  <constructionHeuristicType>BEST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<constructionHeuristic>
  <constructionHeuristicType>WEAKEST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```

- 双方向関係のシャドウ変数では、宣言が **@PlanningVariable(mappedBy)** から **@InverseRelationShadowVariable(sourceVariableName)** に変更になりました。
以前の ***.java** :

```
@PlanningVariable(mappedBy = "previousStandstill")
Customer getNextCustomer();
void setNextCustomer(Customer nextCustomer);
```

現在の ***.java** :

```
@InverseRelationShadowVariable(sourceVariableName = "previousStandstill")
Customer getNextCustomer();
void setNextCustomer(Customer nextCustomer);
```

- 複数の **<planningEntityClass>** 要素では、スーパークラス (およびスーパーインターフェース) が命令される順番が、最後から最初に変更になりました。
以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<planningEntityClass>...TimeWindowedCustomer</planningEntityClass>
<planningEntityClass>...Customer</planningEntityClass>
<planningEntityClass>...Standstill</planningEntityClass>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<planningEntityClass>...Standstill</planningEntityClass>
<planningEntityClass>...Customer</planningEntityClass>
<planningEntityClass>...TimeWindowedCustomer</planningEntityClass>
```

- **<planningEntityClass>** 要素の **<entityClass>** の名前が変更になりました。
以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<planningEntityClass>org.optaplanner.examples.cloudbalancing.domain.CloudProcess</planningEntityClass>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<entityClass>org.optaplanner.examples.cloudbalancing.domain.CloudProcess</entityClass>
```

- **XStreamScoreConverter** および **XStreamBendableScoreConverter** のパッケージが変更になりました。
以前の ***.java** :

```
import org.optaplanner.persistence.xstream.XStreamScoreConverter;
```

現在の ***.java** :

```
import org.optaplanner.persistence.xstream.impl.score.XStreamScoreConverter;
```

以前の ***.java** :

```
import org.optaplanner.persistence.xstream.XStreamBendableScoreConverter;
```

現在の ***.java** :

```
import org.optaplanner.persistence.xstream.impl.score.XStreamBendableScoreConverter;
```

- カスタムの Move を実装している場合は、 **AbstractMove** を展開します。
以前の ***.java** :

```
public class CloudComputerChangeMove implements Move {...}
```

現在の ***.java** :

```
public class CloudComputerChangeMove extends AbstractMove {...}
```

19.2. 値および値の範囲の計画

19.2.1. ValueRangeProvider

収集された数 (たとえば **List<Integer>** または **List<BigDecimal>**) を返す **@ValueRangeProvider** がある場合は、メモリー使用が少なく、さらなる機会を提供できる **ValueRange** に切り替える必要があります。

例:

```
@ValueRangeProvider(id = "delayRange")
public List<Integer> getDelayRange() {
    List<Integer> = new ArrayList<Integer>(5000);
    for (int i = 0; i < 5000; i++) {
        delayRange.add(i);
    }
    return delayRange;
}
```

これは、以下に変更になりました。

```
@ValueRangeProvider(id = "delayRange")
public CountableValueRange<Integer> getDelayRange() {
    return ValueRangeFactory.createIntValueRange(0, 5000);
}
```

注記

アノテーション **@ValueRangeProvider** は、以下のパッケージから移動しました。

```
import org.optaplanner.core.api.domain.value.ValueRangeProvider;
```

以下に変更になりました。

```
import org.optaplanner.core.api.domain.valuerange.ValueRangeProvider;
```

19.2.2. プランニング変数

PlanningVariableListener インターフェースの名前が **VariableListener** に変更になりました。

以前の ***.java** :

```
public class VehicleUpdatingVariableListener implements PlanningVariableListener<Customer> {
```

現在の ***.java** :

```
public class VehicleUpdatingVariableListener implements VariableListener<Customer> {
```

AbstractPlanningVariableListener クラスは削除されました。

以前の ***.java** :

```
public class VehicleUpdatingVariableListener extends AbstractPlanningVariableListener<Customer> {
```

現在の ***.java** :

```
public class VehicleUpdatingVariableListener implements VariableListener<Customer> {
```

VariableListener は、**@PlanningVariable** ではなく、シャドウ変数で宣言されるようになりました。このように、Business Rules Planner はシャドウ変数を認識し、すべてのシャドウ変数は、一貫した内容で宣言されます。さらに、これにより、シャドウ変数が、別のシャドウ変数に基づくことができます。

以前の ***.java** :

```
@PlanningVariable(valueRangeProviderRefs = {"vehicleRange", "customerRange"},
        graphType = PlanningVariableGraphType.CHAINED,
        variableListenerClasses = {VehicleUpdatingVariableListener.class,
ArrivalTimeUpdatingVariableListener.class})
public Standstill getPreviousStandstill() {
    return previousStandstill;
}

public Vehicle getVehicle() {
    return vehicle;
}

public Integer getArrivalTime() {
    return arrivalTime;
}
```

現在の ***.java** :

```
@PlanningVariable(...)
public Standstill getPreviousStandstill() {
    return previousStandstill;
}

@CustomShadowVariable(variableListenerClass = VehicleUpdatingVariableListener.class,
        sources = {@CustomShadowVariable.Source(variableName = "previousStandstill")})
public Vehicle getVehicle() {
    return vehicle;
}

@CustomShadowVariable(variableListenerClass = ArrivalTimeUpdatingVariableListener.class,
        sources = {@CustomShadowVariable.Source(variableName = "previousStandstill")})
public Integer getArrivalTime() {
    return arrivalTime;
}
```

追加設定をしなくても、連鎖変数のアンカーを表わすシャドウ変数に対応するようになりました。たとえば、VRP では、各顧客 (エンティティ) がそれぞれ所有する車 (アンカー) を知る必要があります。この宣言サポートにより、計算を繰り返さなくても、組み込みセレクターがその情報を再利用できます。

以前の ***.java** :

```
@PlanningEntity
public class Customer implements Standstill {

    @PlanningVariable(...)
    public Standstill getPreviousStandstill() {...}

    @CustomShadowVariable(variableListenerClass = VehicleUpdatingVariableListener.class,
            sources = {@CustomShadowVariable.Source(variableName = "previousStandstill")})
    public Vehicle getVehicle() {...}

}
public class VehicleUpdatingVariableListener implements VariableListener<Customer> {
    ...
}
```

現在の ***.java** :

```
@PlanningEntity
public class Customer implements Standstill {

    @PlanningVariable(...)
    public Standstill getPreviousStandstill() {...}

    @AnchorShadowVariable(sourceVariableName = "previousStandstill")
    public Vehicle getVehicle() {...}
}
```

VRP ケースをスケールしなくても、近傍選択は重要です。ようやく完全対応となり、ドキュメントが作成されました。

19.3. ベンチマーク

新しいアグリゲーター機能に対応するために、ベンチマークの内部が完全にリファクタリングされました。

- 「費やす時間 (time spend)」フェーズの名前が「費やした時間 (time spent)」に変更になりました。これには、ログ出力と、ベンチマークレポートが含まれます。

<warmUp*> 要素の名前が変更になりました。

- <warmUpTimeMillisSpend> 要素の名前が <warmUpMillisecondsSpentLimit> に変更になりました。
- <warmUpSecondsSpend> 要素の名前が <warmUpSecondsSpentLimit> に変更になりました。
- <warmUpMinutesSpend> 要素の名前が <warmUpMinutesSpentLimit> に変更になりました。
- <warmUpHoursSpend> 要素の名前が <warmUpHoursSpentLimit> に変更になりました。

以前の ***BenchmarkConfig.xml**

```
<plannerBenchmark>
  <warmUpTimeMillisSpend>...</warmUpTimeMillisSpend>
  <warmUpSecondsSpend>...</warmUpSecondsSpend>
  <warmUpMinutesSpend>...</warmUpMinutesSpend>
  <warmUpHoursSpend>...</warmUpHoursSpend>
  ...
</plannerBenchmark>
```

現在の ***BenchmarkConfig.xml** :

```
<plannerBenchmark>
  <warmUpMillisecondsSpentLimit>...</warmUpMillisecondsSpentLimit>
  <warmUpSecondsSpentLimit>...</warmUpSecondsSpentLimit>
  <warmUpMinutesSpentLimit>...</warmUpMinutesSpentLimit>
  <warmUpHoursSpentLimit>...</warmUpHoursSpentLimit>
  ...
</plannerBenchmark>
```

XmlPlannerBenchmarkFactory クラスは削除され、**PlannerBenchmarkFactory** の静的メソッドに置き換わりました。

以前の ***.java** :

```
PlannerBenchmarkFactory plannerBenchmarkFactory = new XmlPlannerBenchmarkFactory(...);
```

現在の ***.java** :

```
PlannerBenchmarkFactory plannerBenchmarkFactory =
PlannerBenchmarkFactory.createFromXmlResource(...);
```

注記

addXstreamAnnotations() メソッドを使用する場合は、非パブリックの API クラス **XStreamXmlPlannerBenchmarkFactory** に目を通してください。

<xstreamAnnotatedClass> 要素の名前が **<xStreamAnnotatedClass>** に変更になりました。

以前の ***BenchmarkConfig.xml**

```
<problemBenchmarks>

<xstreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</xstreamAnnotated
Class>
  ...
</problemBenchmarks>
```

現在の ***BenchmarkConfig.xml** :

```
<problemBenchmarks>

<xStreamAnnotatedClass>org.optaplanner.examples.nqueens.domain.NQueens</xStreamAnnotated
Class>
  ...
</problemBenchmarks>
```

19.3.1. SolutionFileIO

ProblemIO の名前が **SolutionFileIO** に変更になり、パッケージが変更になりました (パブリック API に追加されました)。

以前の ***.java** :

```
import org.optaplanner.core.impl.solution.ProblemIO;
public class MachineReassignmentFileIO implements ProblemIO {
    ...
}
```

現在の ***.java** :

```
import org.optaplanner.persistence.common.api.domain.solution.SolutionFileIO;
public class MachineReassignmentFileIO implements SolutionFileIO {
    ...
}
```

現在の ***.java** :

以前の ***SolverConfig.xml** および ***BenchmarckConfig.xml** :

```
<problemBenchmarks>
<problemIOClass>...MachineReassignmentProblemIO</problemIOClass>
...
</problemBenchmarks>
```

現在の ***SolverConfig.xml*** および ***BenchmarckConfig.xml*** :

```
<problemBenchmarks>
<solutionFileIOClass>...MachineReassignmentFileIO</solutionFileIOClass>
...
</problemBenchmarks>
```

メソッド **SolutionFileIO.getFileExtension()** が **getInputFileExtension()** と **getOutputFileExtension();** に分かれましたが、ファイル拡張には、入力と出力で同じものを使用することが強く推奨されます。

以前の ***.java** :

```
public String getFileExtension() {
    return FILE_EXTENSION;
}
```

現在の ***.java** :

```
public String getInputFileExtension() {
    return FILE_EXTENSION;
}
public String getOutputFileExtension() {
    return FILE_EXTENSION;
}
```

19.4. SOLVER の設定

Solver および BestSolutionChangedEvent では、**getTimeMillisSpend()** メソッドの名前が **getTimeMillisSpent()** に変更になりました。

以前の ***.java** :

```
... = solver.getTimeMillisSpend();
```

現在の ***.java** :

```
... = solver.getTimeMillisSpent();
```

以前の ***.java** :

```
public void bestSolutionChanged(BestSolutionChangedEvent event) {
    ... = event.getTimeMillisSpend();
}
```

現在の ***.java** :

```
public void bestSolutionChanged(BestSolutionChangedEvent event) {
    ... = event.getTimeMillisSpent();
}
```

Solver フェーズ **<bruteForce>** が、**<exhaustiveSearch>** の **BRUTE_FORCE** タイプに置き換わりました。

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<bruteForce/>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<exhaustiveSearch>
  <exhaustiveSearchType>BRUTE_FORCE</exhaustiveSearchType>
</exhaustiveSearch>
```

setPlanningProblem(Solution) メソッドと **solve()** メソッドが 1 つになり、**solve(Solution)** メソッドとなりました。

以前の ***.java** :

```
solver.setPlanningProblem(planningProblem);
solver.solve();
```

現在の ***.java** :

```
solver.solve(planningProblem);
```

注記

最適解を得るためには、引き続き **solver.getBestSolution()** を使用する必要があります。このメソッドは、今後最適化が必要なくなった場合の対応や、リアルタイムの計画を目的に、意図的に使用されています。

XmlSolverFactory (パブリック API ではない) クラスが削除され、(パブリック API である) **SolverFactory** の静的メソッドに置き換わりました。

以前の ***.java** :

```
SolverFactory solverFactory = new XmlSolverFactory("...solverConfig.xml");
```

現在の ***.java** :

```
SolverFactory solverFactory = SolverFactory.createFromXmlResource("...solverConfig.xml");
```

以前の ***.java** :

```
SolverFactory solverFactory = new XmlSolverFactory().configure(inputStream);
```

現在の ***.java** :

```
SolverFactory solverFactory = SolverFactory.createFromXmlInputStream(inputStream);
```

以前の ***.java** :

```
SolverFactory solverFactory = new XmlSolverFactory().configure(reader);
```

現在の ***.java** :

```
SolverFactory solverFactory = SolverFactory.createFromXmlReader(reader);
```

注記

addXstreamAnnotations() メソッドを使用する場合は、非パブリックの API クラス **XStreamXmlSolverFactory** に目を通してください。

カスタムの SolverPhase に、以下の変更が加えられました。

- **CustomSolverPhaseCommand** インターフェースの名前が **CustomPhaseCommand** に変更になりました。
- **<customSolverPhase>** 要素の名前が **<customPhase>** に変更になりました。
- **<customSolverPhaseCommandClass>** 要素の名前が **>customPhaseCommandClass>** に変更になりました。

以前の ***.java** :

```
public class ToOriginalMachineSolutionInitializer implements CustomSolverPhaseCommand {
    ...
}
```

現在の ***.java** :

```
public class ToOriginalMachineSolutionInitializer implements CustomPhaseCommand {
    ...
}
```

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<customSolverPhase>

<customSolverPhaseCommandClass>...ToOriginalMachineSolutionInitializer</customSolverPhaseCommandClass>
</customSolverPhase>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<customPhase>

<customPhaseCommandClass>....ToOriginalMachineSolutionInitializer</customPhaseCommandClass>
</customPhase>
```

ScoreDefinition.getLevelCount() メソッドの名前が **ScoreDefinition.getLevelsSize()** に変更になりました。

19.5. 最適化

19.5.1. 終了

<termination> の子要素はすべて、名前が変更になりました。

- **<maximumTimeMillisSpend>** 要素の名前が **<millisecondsSpentLimit>** に変更になりました。
- **<maximumSecondsSpend>** 要素の名前が **<secondsSpentLimit>** に変更になりました。
- **<maximumMinutesSpend>** 要素の名前が **<minutesSpentLimit>** に変更になりました。
- **<maximumHoursSpend>** 要素の名前が **<hoursSpentLimit>** に変更になりました。
- **<scoreAttained>** 要素の名前が **<bestScoreLimit>** に変更になりました。
- **<maximumStepCount>** 要素の名前が **<stepCountLimit>** に変更になりました。
- **<maximumUnimprovedStepCount>** 要素の名前が **<unimprovedStepCountLimit>** に変更になりました。

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<termination>
  <maximumTimeMillisSpend>...</maximumTimeMillisSpend>
  <maximumSecondsSpend>...</maximumSecondsSpend>
  <maximumMinutesSpend>...</maximumMinutesSpend>
  <maximumHoursSpend>...</maximumHoursSpend>
  <scoreAttained>...</scoreAttained>
  <maximumStepCount>...</maximumStepCount>
  <maximumUnimprovedStepCount>...</maximumUnimprovedStepCount>
</termination>
```

以下のようになります。

```
<termination>
  <millisecondsSpentLimit>...</millisecondsSpentLimit>
  <secondsSpentLimit>...</secondsSpentLimit>
  <minutesSpentLimit>...</minutesSpentLimit>
  <hoursSpentLimit>...</hoursSpentLimit>
  <bestScoreLimit>...</bestScoreLimit>
  <stepCountLimit>...</stepCountLimit>
  <unimprovedStepCountLimit>...</unimprovedStepCountLimit>
</termination>
```

19.5.2. イベント

BestSolutionChangedEvent クラスおよび **SolverEventListener** クラスが、**impl.event to api.solver.event** パッケージから削除され、パブリック api に追加されました。

以前の ***.java** :

```
import org.optaplanner.core.impl.event.BestSolutionChangedEvent;
import org.optaplanner.core.impl.event.SolverEventListener;
```

現在の ***.java** :

```
import org.optaplanner.core.api.solver.event.BestSolutionChangedEvent;
import org.optaplanner.core.api.solver.event.SolverEventListener;
```

19.5.3. スコアのトレンド

一部のアルゴリズム (構築ヒューリスティック、しらみつぶし探索) のパフォーマンスを向上させるには、**<scoreDirectorFactory>** に **<initializingScoreTrend>** を指定します。

ANY、**ONLY_UP**、および **ONLY_DOWN** を使用する場合は、**InitializingScoreTrend** のドキュメンテーションセクションを参照してください。

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  <scoreDrl>.../cloudBalancingScoreRules.drl</scoreDrl>
</scoreDirectorFactory>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  <scoreDefinitionType>HARD_SOFT</scoreDefinitionType>
  <scoreDrl>.../cloudBalancingScoreRules.drl</scoreDrl>
  <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
</scoreDirectorFactory>
```

<initializingScoreTrend> が指定されていると、 **<constructionHeuristic>** が、最も適切な **<pickEarlyType>** を自動的に使用するため、**<pickEarlyType>** **FIRST_NON_DETERIORATING_SCORE** を **<initializingScoreTrend>** **ONLY_DOWN** に置き換えます。

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  ...
</scoreDirectorFactory>
...
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT_DECREASING</constructionHeuristicType>
  <forager>
    <pickEarlyType>FIRST_NON_DETERIORATING_SCORE</pickEarlyType>
  </forager>
</constructionHeuristic>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  ...
```

```
  <initializingScoreTrend>ONLY_DOWN</initializingScoreTrend>
</scoreDirectorFactory>
...
<constructionHeuristic>
  <constructionHeuristicType>FIRST_FIT_DECREASING</constructionHeuristicType>
</constructionHeuristic>
```

19.5.4. スコア計算プログラム

SimpleScoreCalculator インターフェースの名前が **EasyScoreCalculator** になりました。これにより、別のスコアタイプを返す **SimpleScore** と区別がつきやすくなりました。同様に、パッケージ名も変更になりました。

以前の ***.java** :

```
import org.optaplanner.core.impl.score.director.simple.SimpleScoreCalculator;

public class CloudBalancingEasyScoreCalculator implements SimpleScoreCalculator<CloudBalance>
{
    ...
}
```

現在の ***.java** :

```
import org.optaplanner.core.impl.score.director.easy.EasyScoreCalculator;

public class CloudBalancingEasyScoreCalculator implements EasyScoreCalculator<CloudBalance> {
    ...
}
```

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<simpleScoreCalculatorClass>org.optaplanner.examples.cloudbalancing.solver.score.CloudBalancingEasyScoreCalculator<simpleScoreCalculatorClass>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<easyScoreCalculatorClass>org.optaplanner.examples.cloudbalancing.solver.score.CloudBalancingEasyScoreCalculator
```

BendableScore 設定が変更し、...**LevelCount** の名前が ...**LevelsSize** に変更になりました。

以前の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  <scoreDefinitionType>BENDABLE</scoreDefinitionType>
  <bendableHardLevelCount>2</bendableHardLevelCount>
  <bendableSoftLevelCount>3</bendableSoftLevelCount>
  ...
</scoreDirectorFactory>
```

現在の ***SolverConfig.xml** および ***BenchmarkConfig.xml** :

```
<scoreDirectorFactory>
  <scoreDefinitionType>BENDABLE</scoreDefinitionType>
  <bendableHardLevelsSize>2</bendableHardLevelsSize>
  <bendableSoftLevelsSize>3</bendableSoftLevelsSize>
  ...
</scoreDirectorFactory>
```

第20章 REALTIME DECISION SERVER 機能

20.1. 概要

Realtime Decision Server は、ルールおよびプロセスのインスタンスを作成して実行するのに使用される、モジュール式のスタンドアロンサーバーコンポーネントです。この機能は REST、JMS、および Java の各インターフェースからクライアントアプリケーションに公開されます。

Realtime Decision Server は、主に WAR ファイルとしてパッケージ化される、設定可能な web アプリケーションです。そのディストリビューションは、純粋な web コンテナー (Tomcat など)、JEE 6 コンテナー、および JEE 7 コンテナーで利用できます。

Realtime Decision Server は、拡張の概念に基づいて設定されます。拡張機能は、それぞれ個別に有効または無効にできるため、ニーズに合わせてサーバーを設定できます。

20.2. BUSINESS RESOURCE PLANNER REST API

Planner 機能を有効にすると、Realtime Decision Server は以下に記す追加 REST API に対応します。この API は JMS および Java クライアント API から利用できます。以下の点に注意してください。

- Realtime Decision Server をデプロイするには、『Red Hat JBoss BRMS のユーザーガイド』の「**Realtime Decision Server**」の章を参照してください。
- ベース URL は、上記で定義したエンドポイント (たとえば **http://SERVER:PORT/kie-server/services/rest/server/**) のままになります。
- リクエストするユーザーには、**kie-server** ロールが必要です。
- 特定のマーシャリングフォーマットが必要な場合は、以下のように、HTTP ヘッダー **Content-Type** と、任意で **X-KIE-ContentType** を HTTP リクエストに追加します。

```
Content-Type: application/xml
X-KIE-ContentType: xstream
```

以下の例で使用している要求と応答では、OptaPlanner Workbench の optacloud サンプルを使用して KIE コンテナーを構築していること (以下のように、**/services/rest/server/containers/optacloud-kiecontainer-1** で **PUT** を呼び出す) が前提となっています。

```
<kie-container container-id="optacloud-kiecontainer-1">
  <release-id>
    <group-id>opta</group-id>
    <artifact-id>optacloud</artifact-id>
    <version>1.0.0</version>
  </release-id>
</kie-container>
```

20.2.1. [GET] /containers/{containerId}/solvers

コンテナーで作成した Solver の一覧を返します。

例20.1 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
```

```
<type>SUCCESS</type>
<msg>Solvers list successfully retrieved from container 'optacloud-kiecontainer-1'</msg>
<result class="org.kie.server.api.model.instance.SolverInstanceList">
  <solvers>
    <solver-instance>
      <container-id>optacloud-kiecontainer-1</container-id>
      <solver-id>solver1</solver-id>
      <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
      <status>NOT_SOLVING</status>
    </solver-instance>
    <solver-instance>
      <container-id>optacloud-kiecontainer-1</container-id>
      <solver-id>solver2</solver-id>
      <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
      <status>NOT_SOLVING</status>
    </solver-instance>
  </solvers>
</result>
</org.kie.server.api.model.ServiceResponse>
```

例20.2 サーバーの応答例 (JSON)

```
{
"type" : "SUCCESS",
"msg" : "Solvers list successfully retrieved from container 'optacloud-kiecontainer-1'",
"result" : {
  "solver-instance-list" : {
    "solver" : [ {
      "status" : "NOT_SOLVING",
      "container-id" : "optacloud-kiecontainer-1",
      "solver-id" : "solver1",
      "solver-config-file" : "opta/optacloud/cloudSolverConfig.solver.xml"
    }, {
      "status" : "NOT_SOLVING",
      "container-id" : "optacloud-kiecontainer-1",
      "solver-id" : "solver2",
      "solver-config-file" : "opta/optacloud/cloudSolverConfig.solver.xml"
    } ]
  }
}
}
```

20.2.2. [PUT] /containers/{containerId}/solvers/{solverId}

コンテナー **{containerId}** に、指定の **{solverId}** で新しい Solver を作成します。リクエストの本文は、マーシャリングされた **SolverInstance** エンティティーになり、Solver の設定ファイルに指定する必要があります。

以下は、要求と、それに対応する応答の例です。

例20.3 サーバーのリクエストの例 (XStream)

```
<solver-instance>
  <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
</solver-instance>
```

例20.4 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Solver 'solver1' successfully created in container 'optacloud-kiecontainer-1'</msg>
  <result class="solver-instance">
    <container-id>optacloud-kiecontainer-1</container-id>
    <solver-id>solver1</solver-id>
    <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
    <status>NOT_SOLVING</status>
  </result>
</org.kie.server.api.model.ServiceResponse>
```

例20.5 サーバーのリクエストの例 (JSON)

```
{
  "solver-config-file" : "opta/optacloud/cloudSolverConfig.solver.xml"
}
```

例20.6 サーバーの応答例 (JSON)

```
{
  "type" : "SUCCESS",
  "msg" : "Solver 'solver1' successfully created in container 'optacloud-kiecontainer-1'",
  "result" : {
    "solver-instance" : {
      "container-id" : "optacloud-kiecontainer-1",
      "solver-id" : "solver1",
      "solver-config-file" : "opta/optacloud/cloudSolverConfig.solver.xml",
      "status" : "NOT_SOLVING"
    }
  }
}
```

20.2.3. [GET] /containers/{containerId}/solvers/{solverId}

コンテナー **{containerId}** で Solver **{solverId}** に関する現在のステータスを返します。

例20.7 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Solver 'solver1' state successfully retrieved from container 'optacloud-kiecontainer-
```

```
1'</msg>
  <result class="solver-instance">
    <container-id>optacloud-kiecontainer-1</container-id>
    <solver-id>solver1</solver-id>
    <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
    <status>NOT_SOLVING</status>
  </result>
</org.kie.server.api.model.ServiceResponse>
```

例20.8 サーバーの応答例 (JSON)

```
{
  "type" : "SUCCESS",
  "msg" : "Solver 'solver1' state successfully retrieved from container 'optacloud-kiecontainer-1'",
  "result" : {
    "solver-instance" : {
      "container-id" : "optacloud-kiecontainer-1",
      "solver-id" : "solver1",
      "solver-config-file" : "opta/optacloud/cloudSolverConfig.solver.xml",
      "status" : "NOT_SOLVING"
    }
  }
}
```

20.2.4. 解決の開始

以下は、2 台のコンピューターと 1 つのプロセスで発生した optacloud 問題を解決する例です。

例20.9 サーバーのリクエストの例 (XStream)

```
<solver-instance>
  <status>SOLVING</status>
  <planning-problem class="opta.optacloud.CloudSolution">
    <computerList>
      <opta.optacloud.Computer>
        <cpuPower>10</cpuPower>
        <memory>4</memory>
        <networkBandwidth>100</networkBandwidth>
        <cost>1000</cost>
      </opta.optacloud.Computer>
      <opta.optacloud.Computer>
        <cpuPower>20</cpuPower>
        <memory>8</memory>
        <networkBandwidth>100</networkBandwidth>
        <cost>3000</cost>
      </opta.optacloud.Computer>
    </computerList>
    <processList>
      <opta.optacloud.Process>
        <requiredCpuPower>1</requiredCpuPower>
        <requiredMemory>7</requiredMemory>
        <requiredNetworkBandwidth>1</requiredNetworkBandwidth>
```

```
      </opta.optacloud.Process>
    </processList>
  </planning-problem>
</solver-instance>
```

応答には、最適解が含まれていません。解決するには数秒、数分、数時間、または数日かかる場合があり、HTTP 要求がタイムアウトになるためです。

例20.10 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Solver 'solver1' from container 'optacloud-kiecontainer-1' successfully updated.</msg>
  <result class="solver-instance">
    <container-id>optacloud-kiecontainer-1</container-id>
    <solver-id>solver1</solver-id>
    <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
    <status>SOLVING</status>
  </result>
</org.kie.server.api.model.ServiceResponse>
```

代わりに、解決は非同期に行われるため、最適解を得るには、**bestsolution** URL を呼び出す必要があります。

20.2.5. 解決の終了

たとえば、解決を終了するには、以下のようにします。

例20.11 サーバーのリクエストの例 (XStream)

```
<solver-instance>
  <status>NOT_SOLVING</status>
</solver-instance>
```

例20.12 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Solver 'solver1' from container 'optacloud-kiecontainer-1' successfully updated.</msg>
  <result class="solver-instance">
    <container-id>optacloud-kiecontainer-1</container-id>
    <solver-id>solver1</solver-id>
    <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
    <status>TERMINATING_EARLY</status>
    <score class="org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore">
      <hardScore>0</hardScore>
      <softScore>-3000</softScore>
    </score>
  </result>
</org.kie.server.api.model.ServiceResponse>
```

-

Solver は削除されないため、終了しても最適解は取得できます。

20.2.6. [GET] /containers/{containerId}/solvers/{solverId}/bestsolution

リクエストの作成時に見つかった最適解を返します。Solver は終了していないため (つまり **status** フィールドは **SOLVING** のままなので)、返される最適解は、その時点での最適解であり、後で別の解が返される可能性もあります。

たとえば、上述の問題は、以下の解 (ソリューション) を返します。ここでは、(1 台目のコンピューターではメモリーが不足しているため) 2 台目のコンピューターにプロセスが割り当てられています。

例20.13 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Best computed solution for 'solver1' successfully retrieved from container 'optacloud-
kiecontainer-1'</msg>
   <result class="solver-instance">
    <container-id>optacloud-kiecontainer-1</container-id>
    <solver-id>solver1</solver-id>
    <solver-config-file>opta/optacloud/cloudSolverConfig.solver.xml</solver-config-file>
    <status>SOLVING</status>
    <score class="org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore">
      <hardScore>0</hardScore>
      <softScore>-3000</softScore>
    </score>
    <best-solution class="opta.optacloud.CloudSolution">
      <score class="org.optaplanner.core.api.score.buildin.hardsoft.HardSoftScore"
reference="../../score" />
      <computerList>
        <opta.optacloud.Computer>
          <cpuPower>10</cpuPower>
          <memory>4</memory>
          <networkBandwidth>100</networkBandwidth>
          <cost>1000</cost>
        </opta.optacloud.Computer>
        <opta.optacloud.Computer>
          <cpuPower>20</cpuPower>
          <memory>8</memory>
          <networkBandwidth>100</networkBandwidth>
          <cost>3000</cost>
        </opta.optacloud.Computer>
      </computerList>
      <processList>
        <opta.optacloud.Process>
          <requiredCpuPower>1</requiredCpuPower>
          <requiredMemory>7</requiredMemory>
          <requiredNetworkBandwidth>1</requiredNetworkBandwidth>
          <computer reference="../../../computerList/opta.optacloud.Computer[2]" />
        </opta.optacloud.Process>
      </processList>
```

```
    </best-solution>
  </result>
</org.kie.server.api.model.ServiceResponse>
```

20.2.7. [DELETE] /containers/{containerId}/solvers/{solverId}

コンテナー **{containerId}** の Solver **{solverId}** を破棄します。Solver が終了していない場合は、終了してから行います。

例20.14 サーバーの応答例 (XStream)

```
<org.kie.server.api.model.ServiceResponse>
  <type>SUCCESS</type>
  <msg>Solver 'solver1' successfully disposed from container 'optacloud-kiecontainer-1'</msg>
</org.kie.server.api.model.ServiceResponse>
```

例20.15 サーバーの応答例 (JSON)

```
{
  "type" : "SUCCESS",
  "msg" : "Solver 'solver1' successfully disposed from container 'optacloud-kiecontainer-1'"
}
```

付録A バージョン情報

Documentation last updated on: Wednesday, Oct 23, 2019.