Оглавление

3 Задачи с ограничениями

Многие задачи, для решения которых используются компьютерные вычисления, можно в целом отнести к категории задач с ограничениями (constraint-satisfaction problems, CSP). CSP-задачи состоят из *переменных*, допустимые значения которых попадают в определенные диапазоны, известные как *области определения*. Для того чтобы решить задачу с ограничениями, необходимо удовлетворить существующие ограничения для переменных. Три основных понятия — переменные, области определения и ограничения — просты и понятны, а благодаря их универсальности задачи с ограничениями получили широкое применение.

Рассмотрим пример такой задачи. Предположим, что вы пытаетесь назначить на пятницу встречу для Джо, Мэри и Сью. Сью должна встретиться хотя бы с одним человеком. В этой задаче планирования переменными могут быть три человека — Джо, Мэри и Сью. Областью определения для каждой переменной могут быть часы, когда свободен каждый из них. Например, у переменной «Мэри» область определения составляет 2, 3 и 4 часа пополудни. У этой задачи есть также два ограничения. Во-первых, Сью должна присутствовать на встрече. Во-вторых, на встрече должны присутствовать по крайней мере два человека. Решение этой задачи с ограничениями определяется тремя переменными, тремя областями определения и двумя ограничениями, тогда задача будет решена и при этом не придется объяснять пользователю, *как именно* (рис. 3.1).

В некоторых языках программирования, таких как Prolog и Picat, есть встроенные средства для решения задач с ограничениями. В других языках обычным подходом является создание структуры, которая включает в себя поиск с возвратами и несколько эвристик для повышения производительности поиска. В этой главе мы сначала создадим структуру для CSP-задач, которая будет решать их простым рекурсивным поиском с возвратами. Затем воспользуемся этой структурой для решения нескольких примеров таких задач.

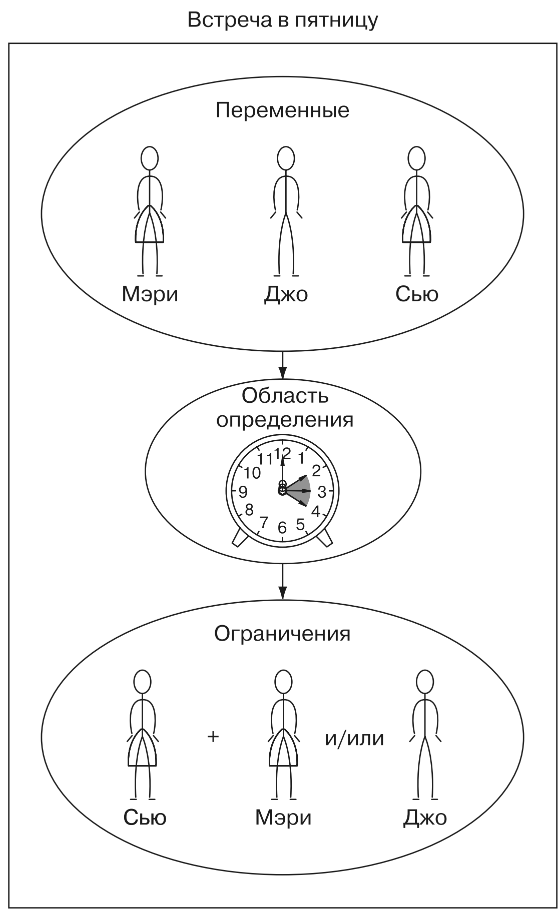

Рис. 3.1. Задачи планирования — это классическое применение структур для удовлетворения ограничений

3.1. ПОСТРОЕНИЕ СТРУКТУРЫ ДЛЯ ЗАДАЧИ С ОГРАНИЧЕНИЯМИ

Определим ограничения посредством класса `Constraint`. Каждое ограничение `Constraint` состоит из переменных `variables`, которые оно ограничивает, и метода `satisfied()`, который проверяет, выполняется ли оно. Определение того, выполняется ли ограничение, является основной логикой, входящей в определение конкретной задачи с ограничениями. Реализацию по умолчанию нужно переопределить. Именно так и должно быть, потому что мы определяем `Constraint` как абстрактный базовый класс. Абстрактные базовые классы не предназначены для реализации. Только их подклассы, которые переопределяют и реализуют свои абстрактные методы `abstract`, предназначены для действительного использования (листинг 3.1).

Листинг 3.1. Constraint.java

```
package chapter3;

import java.util.List;
import java.util.Map;

// V — тип переменной, D — тип домена.
public abstract class Constraint<V, D> {

    // Переменные, для которых существует ограничение
    protected List<V> variables;

    public Constraint(List<V> variables) {
        this.variables = variables;
    }

    // Необходимо переопределить в подклассах
    public abstract boolean satisfied(Map<V, D> assignment);
}
```

СОВЕТ

В Java бывает сложно выбрать между абстрактным классом и интерфейсом. Абстрактные классы могут содержать переменные экземпляра. Поскольку у нас есть переменные экземпляра переменных, мы используем абстрактный класс.

Центральным элементом нашей структуры соответствия ограничениям будет класс с названием `CSP` (листинг 3.2). `CSP` — это место, где собраны все переменные, области определения и ограничения. С точки зрения подсказок типов класс `CSP` применяет универсальные средства, чтобы быть достаточно гибким, работать с любыми значениями переменных и областей определения (где `V` — это значения переменных, а `D` — значения областей определения). В `CSP` коллекции `variables`, `domains` и `constraints` имеют ожидаемые типы. Коллекция `variables` — это `list` для переменных, `domains` — `Map` с соответствием переменных спискам возможных значений (областям определения этих переменных), а `constraints` — `Map`, где каждой переменной соответствует `list` наложенных на нее ограничений.

Конструктор создает карту `constraints Map`. Метод `addConstraint()` просматривает все переменные, к которым относится данное ограничение, и добавляет себя в соответствие `constraints` для каждой такой переменной. Оба метода имеют простейшую проверку ошибок и вызывают исключение, если `variable` отсутствует в области определения или существует `constraint` для несуществующей переменной.

Листинг 3.2. CSP.java (продолжение)

```
package chapter3;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CSP<V, D> {
    private List<V> variables;
    private Map<V, List<D>> domains;
    private Map<V, List<Constraint<V, D>>> constraints = new HashMap<>();

    public CSP(List<V> variables, Map<V, List<D>> domains) {
        this.variables = variables;
        this.domains = domains;
        for (V variable : variables) {
            constraints.put(variable, new ArrayList<>());
            if (!domains.containsKey(variable)) {
                throw new IllegalArgumentException("Every variable should
                      have a domain assigned to it.");
            }
        }
    }

public void addConstraint(Constraint<V, D> constraint) {
    for (V variable : constraint.variables) {
        if (!variables.contains(variable)) {
            throw new IllegalArgumentException("Variable in constraint
                  not in CSP");
        }
        constraints.get(variable).add(constraint);
    }
}
```

Как узнать, соответствует ли данная конфигурация переменных и выбранных значений области определения заданным ограничениям? Мы будем называть такую заданную конфигурацию *присваиванием*. Нам нужна функция, которая проверяла бы каждое ограничение для заданной переменной по отношению к присваиванию, чтобы увидеть, удовлетворяет ли значение переменной в присваивании этим ограничениям. В листинге 3.3 реализована функция `consistent()` как метод класса `CSP`.

Метод `consistent()` перебирает все ограничения для данной переменной (это всегда будет переменная, только что добавленная в присваивание) и проверяет, выполняется ли ограничение, учитывая новое присваивание. Если присваивание удовлетворяет всем ограничениям, возвращается `True`. Если какое-либо ограничение, наложенное на переменную, не выполняется, возвращается значение `False`.

Листинг 3.3. CSP.java (продолжение)

```
// Проверяем, соответствует ли присваивание значения,
// проверяя все ограничения для данной переменной
public boolean consistent(V variable, Map<V, D> assignment) {
    for (Constraint<V, D> constraint : constraints.get(variable)) {
        if (!constraint.satisfied(assignment)) {
            return false;
        }
    }
    return true;
}
```

Для поиска решения задачи в такой структуре выполнения ограничений будет использоваться простой поиск с возвратами. *Возвраты* — это подход, при котором, если поиск зашел в тупик, мы возвращаемся к последней известной точке, где было принято решение, перед тем как зайти в тупик, и выбираем другой путь. Если вам кажется, что это похоже на поиск в глубину из главы 2, то вы правы. Поиск с возвратом, реализованный в функции `backtrackingSearch()`, — это своего рода рекурсивный поиск в глубину, в котором объединены идеи, описанные в главах 1 и 2. Эта функция добавляется в качестве метода в класс `CSP` (листинг 3.4).

Листинг 3.4. CSP.java (продолжение)

```
public Map<V, D> backtrackingSearch(Map<V, D> assignment) {
    // присваивание завершено, если существует присваивание
    // для каждой переменной (базовый случай)
    if (assignment.size() == variables.size()) {
        return assignment;
    }
    // получить все переменные из CSP, но не из присваивания
    V unassigned = variables.stream().filter(v ->
      !assignment.containsKey(v)).findFirst().get();
    // получить все возможные значения области определения
    // для первой переменной без присваивания
    for (D value : domains.get(unassigned)) {
        // мелкая копия присваивания, которую мы можем изменить
        Map<V, D> localAssignment = new HashMap<>(assignment);
        localAssignment.put(unassigned, value);
        // если нет противоречий, продолжаем рекурсию
        if (consistent(unassigned, localAssignment)) {
            Map<V, D> result = backtrackingSearch(localAssignment);
            // если результат не найден, заканчиваем возвраты
            if (result != null) {
                return result;
            }
        }
```

```
        }
        return null;
    }

    // вспомогательный класс функции backtrackingSearch
    public Map<V, D> backtrackingSearch() {
        return backtrackingSearch(new HashMap<>());
    }
}
```

Исследуем `backtrackingSearch()` построчно.

```
if (assignment.size() == variables.size()) {
    return assignment;
}
```

Базовый случай для рекурсивного поиска означает, что нужно найти правильное присваивание для каждой переменной. Сделав это, мы возвращаем первый валидный экземпляр решения (и не продолжаем поиск).

```
V unassigned = variables.stream().filter(v ->
    !assignment.containsKey(v)).findFirst().get();
```

Чтобы выбрать новую переменную, область определения которой будем исследовать, мы просто просматриваем все переменные и находим первую, которая не имеет присваивания. Для этого создаем поток данных, отфильтрованных по `assignment`, и извлекаем первую, которая не назначена, с помощью `findFirst().filter()`, принимая `Predicate`. `Predicate` — это функциональный интерфейс, описывающий функцию, который принимает один аргумент и возвращает логическое значение. Предикат — это лямбда-выражение (`v -> !Assignment.containsKey(v)`), которое возвращает значение `true`, если `assignment` не содержит аргумент, который в данном случае будет переменной для нашего `CSP`.

```
for (D value : domains.get(unassigned)) {
    Map<V, D> localAssignment = new HashMap<>(assignment);
    localAssignment.put(unassigned, value);
```

Если новое присваивание в `localAssignment` согласуется со всеми ограничениями, что проверяется с помощью `consistent()`, мы продолжаем рекурсивный поиск для нового присваивания. Если новое присваивание оказывается завершенным (базовый случай), передаем его вверх по цепочке рекурсии.

```
if (consistent(unassigned, localAssignment)) {
    Map<V, D> result = backtrackingSearch(localAssignment);
    if (result != null) {
        return result;
    }
}
```

Для данной переменной мы пытаемся определить все возможные значения домена. Каждое новое значение будет храниться на локальной карте с именем `localAssignment`.

```
return null;
```

Наконец, если мы рассмотрели все возможные значения области определения для конкретной переменной и не обнаружили решения, в котором использовался бы существующий набор назначений, то возвращаем `null`, что указывает на отсутствие решения. В результате по цепочке рекурсии будет выполнен возврат к точке, в которой могло быть принято другое предварительное присваивание.

3.2. ЗАДАЧА РАСКРАШИВАНИЯ КАРТЫ АВСТРАЛИИ

Представьте, что у вас есть карта Австралии, на которой вы хотите разными цветами обозначить штаты/территории (мы будем называть те и другие регионами). Никакие две соседние области не должны быть окрашены в один и тот же цвет. Можно ли раскрасить регионы всего тремя разными цветами?

Ответ: да. Попробуйте сами (самый простой способ — напечатать карту Австралии с белым фоном). Мы, люди, можем быстро найти решение путем изучения карты и небольшого количества проб и ошибок. На самом деле это тривиальная задача, которая отлично подойдет в качестве первой для нашей программы решения задач с ограничениями методом поиска с возвратом (рис. 3.2).

Чтобы смоделировать проблему как CSP, нужно определить переменные, области определения и ограничения. Переменные — это семь регионов Австралии (по крайней мере те семь, которыми мы ограничимся): Западная Австралия, Северная территория, Южная Австралия, Квинсленд, Новый Южный Уэльс, Виктория и Тасмания. В нашем CSP их можно представить как строки. Область определения каждой переменной — это три разных цвета, которые могут быть ей присвоены. (Мы используем красный, зеленый и синий.) Ограничения — сложный вопрос. Никакие две соседние области не могут быть окрашены в один и тот же цвет, поэтому ограничения будут зависеть от того, какие из них граничат друг с другом. Мы задействуем так называемые двоичные ограничения — ограничения между двумя переменными. Каждые две области с общей границей будут иметь двоичное ограничение, указывающее, что им нельзя присвоить один и тот же цвет.

Чтобы реализовать такие двоичные ограничения в коде, нужно создать подкласс класса `Constraint`. Конструктор подкласса `MapColoringConstraint` будет принимать две переменные — две области, имеющие общую границу. Его переопределенный метод `satisfied()` сначала проверит, присвоены ли этим двум областям значения (цвета) из области определения. Если нет, то ограничение считается тривиально выполненным до тех пор, пока цвета не будут присвоены. (Пока у одного из регионов нет цвета, конфликт невозможен.) Затем метод проверит, присвоен ли двум областям один и тот же цвет. Очевидно, что если цвета одинаковы, то существует конфликт, означающий: ограничение не выполняется.

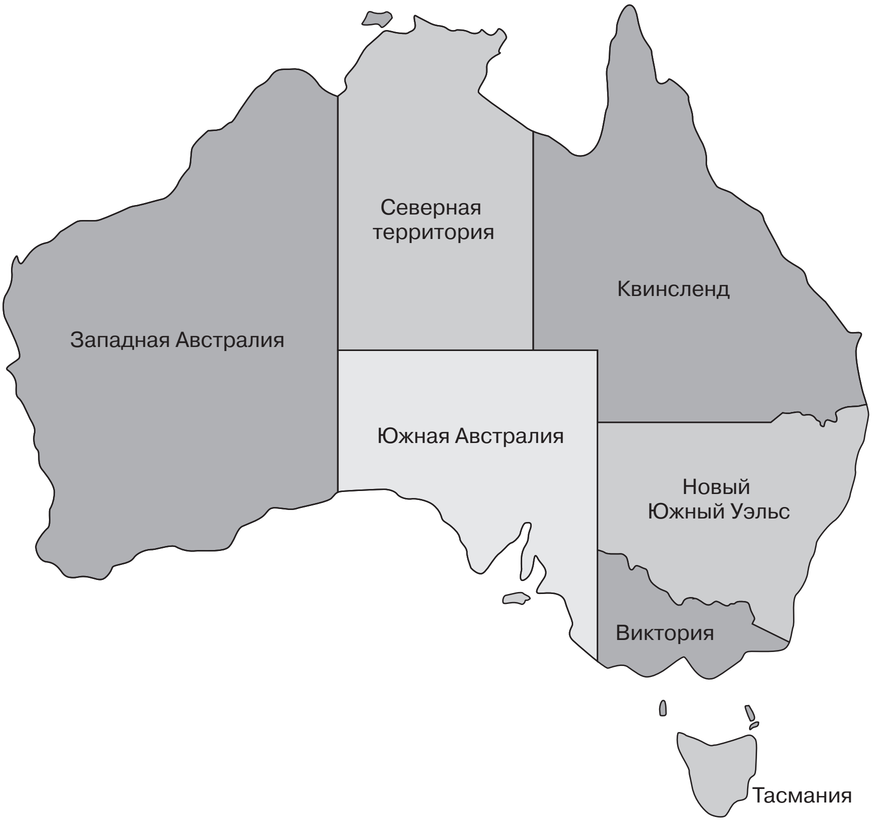

Рис. 3.2. При раскраске карты Австралии никакие два смежных региона не могут быть окрашены в один и тот же цвет

Далее этот класс представлен во всей своей полноте (листинг 3.5). Сам по себе класс `MapColoringConstraint` не универсален с точки зрения аннотации типа, но он является подклассом параметризованной версии универсального класса `Constraint`, которая указывает, что переменные и области определения имеют тип `String`.

Листинг 3.5. MapColoringConstraint.java

```
package chapter3;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public final class MapColoringConstraint extends Constraint<String, String> {
    private String place1, place2;

    public MapColoringConstraint(String place1, String place2) {
        super(List.of(place1, place2));
        this.place1 = place1;
        this.place2 = place2;
    }

    @Override
    public boolean satisfied(Map<String, String> assignment) {
        // если какой-либо регион place отсутствует в присваивании,
        // то его цвета не могут привести к конфликту
        if (!assignment.containsKey(place1) ||
            !assignment.containsKey(place2)) {
            return true;
        }
        // проверяем, не совпадает ли цвет, присвоенный
        // place1, с цветом, присвоенным place2
        return !assignment.get(place1).equals(assignment.get(place2));
    }
```

Теперь, когда есть способ реализации ограничений между регионами, можно легко уточнить задачу о раскрашивании карты Австралии с помощью программы решения методом CSP — нужно лишь заполнить области определения и переменные, а затем добавить ограничения (листинг 3.6).

Листинг 3.6. MapColoringConstraint.java (продолжение)

```
    public static void main(String[] args) {
        List<String> variables = List.of("Western Australia", "Northern
            Territory", "South Australia", "Queensland", "New South Wales",
            "Victoria", "Tasmania");
        Map<String, List<String>> domains = new HashMap<>();
        for (String variable : variables) {
            domains.put(variable, List.of("red", "green", "blue"));
        }
        CSP<String, String> csp = new CSP<>(variables, domains);
        csp.addConstraint(new MapColoringConstraint("Western Australia",
            "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Western Australia",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("South Australia",
            "Northern Territory"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "Northern Territory"));
```

```
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Queensland",
            "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("New South Wales",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
            "South Australia"));
        csp.addConstraint(new MapColoringConstraint("Victoria",
            "New South Wales"));
        csp.addConstraint(new MapColoringConstraint("Victoria", "Tasmania"));
```

Наконец, вызываем `backtrackingSearch()` для поиска решения (листинг 3.7).

Листинг 3.7. MapColoringConstraint.java (продолжение)

```
        Map<String, String> solution = csp.backtrackingSearch();
        if (solution == null) {
            System.out.println("No solution found!");
        } else {
            System.out.println(solution);
        }
    }

}
```

Правильное решение будет представлять собой список цветов, присвоенных регионам:

```
{Western Australia=red, New South Wales=green, Victoria=red, Tasmania=green,
  Northern Territory=green, South Australia=blue, Queensland=red}
```

3.3. ЗАДАЧА ВОСЬМИ ФЕРЗЕЙ

Шахматная доска — это сетка размером 8 × 8 клеток. Ферзь — шахматная фигура, которая может перемещаться по шахматной доске на любое количество клеток по любой горизонтали, вертикали или диагонали. Если за один ход ферзь может переместиться на клетку, на которой стоит другая фигура, не перепрыгивая ни через какую другую фигуру, то ферзь атакует эту фигуру. (Другими словами, если фигура находится в зоне прямой видимости ферзя, то она подвергается атаке.) Задача восьми ферзей состоит в том, как разместить восемь ферзей на шахматной доске таким образом, чтобы ни один из них не атаковал другого (рис. 3.3).

Чтобы представить клетки на шахматной доске, присвоим каждой из них два целых числа, обозначающих горизонталь и вертикаль. Мы можем гарантировать, что никакая пара из восьми ферзей не находится на одной вертикали, просто присвоив им последовательно номера вертикалей с первого по восьмой.