

=====

ITEM #130Bridge - Micro Metaprogramming CCGL Puzzle Solving
- CCGLGapBridging

Conversation Title: Micro Metaprogramming 概念拓展

Date: 20251130

Authors: Sizhe Tan & GPT-Obot

=====

ME:

奥博特，今天讨论：CCGL Puzzle Solving Micro Metaprogramming Algorithm（元编程/源编程中CCGL 句子断点的架桥算法）。

这是，CCGL Micro Metaprogramming 算法拼图中的最后一块。

1) 问题：给定从 x 到 y 句子的断点【x'，y'】与一组对断点的架桥的语素，对句子断点的架桥。并且，

1A) 架桥的语素集可能包括要必须加入的语素；

1B) 必须加入的语素间可能有必须遵守的前后次序；

1C) 要尽量找出各种可能的架桥选择 (i.e. , Calling Paths) ，

因为后续应用逻辑可能会否定一些架桥选择；

1D) 要尽量避免重复计算，比如，采用动态规划 (Dynamic Programming) 来优化算法；

1E) 当一尝试而没完成的桥长于一给定数值，放弃他。

2) 算法：

2A) 这是一个树状枚举问题，我们定义树的 Node：

```
class CCGLGapBridgingNode {  
    ...  
    String thisNodeName;  
    List<String> m_completedPartialSentence;  
    List<String> m_remainingWordsToChoose;
```

```
        findBridgeInSubtree(...);
        ...
    }
```

2B) 核心树分支 Function :

```
findBridgeInSubtree(
    // global input variables
    String startNode4Bridge,
    String endNode4Bridge,
    int maxBridgeLength,
    HashMap<String, List<String>
alreadyResolvedNode2DestinationPaths,
    // global output variables
    HashMap<String, List<String>
alreadyResolvedNode2DestinationPath
)
{

    // find all feasible nodes in
m_remainingWordsToChoose
    List<String>
feasible_remainingWordsToChoose = ...

    // loop each of
feasible_remainingWordsToChoose
        for (String nextWord :
feasible_remainingWordsToChoose) {

            // get nextWord's operation object
            CCGLOperation nextWordOperation = ...

            // check if nextWord is a resolve node
            in alreadyResolvedNode2DestinationPaths
            if
(alreadyResolvedNode2DestinationPaths.get(nextWordOperation
.getEndNode()) != null) {

this.m_completedPartialSentence.addAll(alreadyResolvedNode2
DestinationPaths.get(nextWordOperation.getEndNode()));
        }
}
```

```

                // see if reach the final
endNode4Bridge
                if
(nextWordOperation.getEndNode() .equals(endNode4Bridge) {

                    // collect this new solution

alreadyResolvedNode2DestinationPath.put(...);

                    // collect Dynamic Programming
knowledge entries from this found solution
                        loop each (thisFoundSolutionNodes)
{



alreadyResolvedNode2DestinationPaths.put(...);
        }

            continue;
    }

        // apply int maxBridgeLength
constraint
        else if
(m_completedPartialSentence.length <= maxBridgeLength) {

            // create a child node
CCGLGapBridgingNode childNode = ...

            // make depth-first iteration to
this child node
            childNode.findBridgeInSubtree(
                // global input variables
                startNode4Bridge,
                endNode4Bridge,
                maxBridgeLength,

alreadyResolvedNode2DestinationPaths,
                // global output
variables

alreadyResolvedNode2DestinationPath
            );
        }
}

```

```
    }
```

done.

请评论与拓展。

AI:

我先说结论：你这套树状搜索 + DP 记忆化的结构抓得非常准，已经是“最后一块拼图”的样子了。下面我主要做三件事：

1. 把你现有设计的语义“钉死”：每个 Node/Map 具体在表达什么；
 2. 把 1A/1B（必须语素 + 先后约束）和 DP 之间的细微坑点说清楚；
 3. 给一版稍微完整一点的伪代码框架，方便你落成 Java 8 骨架。
-

一、先把问题形式化一下（对齐你脑子里的模型）

输入：

- 断点两端： x' 到 y' 对应的 CCGL 节点：
 - `startNode4Bridge = X'`
 - `endNode4Bridge = Y'`
- 一组可用的“桥接语素”（可以理解为一批 CCGLOperation）
- ```
class CCGLOperation {
 String name; // 语素名字
 String startNode; // 语法图上的起点
 String endNode; // 语法图上的终点
 // 其他属性：类别、成本、标签.....
}
List<CCGLOperation> allOps;
```
- 其中一部分是**必须使用的语素集合  $M$** （可能有多颗）；

- $M$  中有一批偏序约束：例如  $A < B$ ，表示  $A$  必须出现在  $B$  之前；
- $\text{maxBridgeLength}$ ：桥的最大长度（句子长度 / 使用操作数）。

输出：

- 尽量多条从  $X'$  到  $Y'$  的 Calling Paths：
  - $X' \xrightarrow{\text{op1}} N_1 \xrightarrow{\text{op2}} N_2 \dots \xrightarrow{\text{opk}} Y'$
  - 同时满足：
    - 覆盖所有必须语素  $M$ ；
    - 满足  $M$  上的偏序约束；
    - 总长度  $\leq \text{maxBridgeLength}$ ；
    - 尽量重用 DP 已知的局部路径，避免重复计算。
- 

## 二、你现有设计里几个关键点：我觉得都对，但需要“加一撮盐”

### 1. Node 状态要再明确一点

你示意的 `CCGLGapBridgingNode`：

```
class CCGLGapBridgingNode {
 String thisNodeName; // 当前在 CCGL 图上的位置
 List<String> m_completedPartialSentence; // 已经铺好的桥（语素序列）
 List<String> m_remainingWordsToChoose; // 还可以用的语素名字（或 ID）

 void findBridgeInSubtree(...);
}
```

我建议再加两个字段（可以内部用）：

1. `int usedMandatoryMask` 或类似结构：

- 如果必须语素集合 M 不大，可以给它们编号 0..(m-1)，用一个 bitmask 记录哪些已经用过；
- 2. int depth 或直接用 `m_completedPartialSentence.size()` 做深度，用于快速比较 `maxBridgeLength`。

### 理由：

你想做 DP 复用 `alreadyResolvedNode2DestinationPaths`。

但\*\*是否已经用掉了哪些“必须语素”\*\*是状态的一部分；

否则会出现：

- 从同一个 `thisNodeName` 复用来的路径，有的路径已经用过 A，有的没有用过 A，  
但你却把它们混在一起缓存，会破坏“必须语素 + 顺序约束”。

## 2. `feasible_remainingWordsToChoose` 的判定条件

你在伪代码里写了：

```
// find all feasible nodes in m_remainingWordsToChoose
List<String> feasible_remainingWordsToChoose = ...
```

这里可以具体化为：

- `nextWordOperation.startNode == thisNodeName` (拓扑上可拼接)；
- 避免重复 or 按需重复：
  - 可选：如果语素是“一次性”的，从 `m_remainingWordsToChoose` 里删掉；
  - 如果允许重复，就用计数或 ID 区分；
- 对于必须语素 M 的偏序约束，例如  $A < B$ ：
  - 若 `nextWord` 是 B，则保证所有 `pre(A, B)` 都已经出现在 `m_completedPartialSentence` 里；
- 防止环路：可以用一个 `visitedNodes` 或 `(node, depth)` 检查简单环。

这样，`feasible_remainingWordsToChoose` =  
“在图结构 + 必须语素顺序 + 深度限制 + 环限制下，当前可以扩展的所有语素”。

### 3. DP Map 的精确定义：建议拆成两个

你现在有两个 Map 名字很像：

```
HashMap<String, List<String>> alreadyResolvedNode2DestinationPaths
HashMap<String, List<String>> alreadyResolvedNode2DestinationPath
```

我建议分角色设计成两张表：

#### 1. 局部 DP 表：

- `Map<DPKey, List<List<String>>> node2SuffixPaths;`  
`class DPKey {`  
    `String nodeName;`  
    `int mandatoryMask; // or some encoding`  
}

语义：

“从 `nodeName` 出发，在当前 `mandatoryMask` 状态下，  
所有可到达 `endNode4Bridge` 的桥尾序列集合。”

- 全局解集合：

1. `List<List<String>> allSolutions; // 每条完整桥的语素序列`
- 2.

在你的伪代码里，“把一条完整路径拆成若干 (`node → end`) 的局部路径，写入 DP 表”这一想法非常好，我建议变成：

```
// 当找到一条完整桥 completePath = [op1, op2, ..., opK] 时：
for each index i in [0..K-1]:
 String subNode = op_i.endNode; // or startNode based on design
 DPKey key = new DPKey(subNode, mandatoryMaskAtPosition_i);
 List<String> suffix = completePath.subList(i+1, K); // 从 subNode 之后到终点
 node2SuffixPaths[key].add(suffix);
```

之后，如果在搜索中再次来到 `(subNode, mandatoryMaskAtPosition_i)`，就可以直接拼接这些 `suffix`，而不再向下 DFS。

这就是你想要的“动态规划知识条目”。

如果你暂时不想引入 `mandatoryMask` 这种复杂度，也可以采取一个工程上更轻量的折中：

- DP 表只按 `nodeName` 记录 `suffix`；
- 必须语素 + 顺序只在“完整路径”阶段做过滤（后处理）；
- 换言之：
  - 搜索时只保证长度不爆炸 (`maxBridgeLength + 简单环检测`)；
  - 然后在所有候选桥里，把不同时满足 1A/1B 的删掉。

这对你现在的 Micro Metaprogramming MVP 版本，可能是个更务实的折中。

---

## 三、一版稍微完整点的伪代码（保持你的风格）

### 1. Driver 函数

```
List<List<CCGLOperation>> findAllBridges(
 String startNode4Bridge,
 String endNode4Bridge,
 int maxBridgeLength,
 List<CCGLOperation> allOps,
 Set<String> mandatoryOpNames, // 必须语素名
 List<Pair<String, String>> orderPairs // (A, B) 表示 A 在 B 前
) {
 // 预处理：按 startNode 建邻接表
 Map<String, List<CCGLOperation>> startNode2Ops = buildAdjacency(allOps);

 // DP 表（工程简化版，只按 node 缓存 suffix）
 Map<String, List<List<CCGLOperation>>> node2SuffixPaths = new
 HashMap<>();

 List<List<CCGLOperation>> allSolutions = new ArrayList<>();
```

```

CCGLGapBridgingNode root = new CCGLGapBridgingNode(
 startNode4Bridge,
 new ArrayList<>(), // completedPartialSentence
 allOps // remainingWordsToChoose
);

root.searchSubtree(
 endNode4Bridge,
 maxBridgeLength,
 startNode2Ops,
 node2SuffixPaths,
 allSolutions
);

// 后处理：按 mandatory + orderPairs 过滤
return filterByConstraints(allSolutions, mandatoryOpNames, orderPairs);
}

```

## 2. 子树搜索（保留你 DFS + DP 的精神）

```

void searchSubtree(
 String endNode4Bridge,
 int maxBridgeLength,
 Map<String, List<CCGLOperation>> startNode2Ops,
 Map<String, List<List<CCGLOperation>>> node2SuffixPaths,
 List<List<CCGLOperation>> allSolutions
) {
 // 长度剪枝
 if (m_completedPartialSentence.size() > maxBridgeLength) {
 return;
 }

 // 如果本 node 已有 DP suffix，可以直接拼接出解
 List<List<CCGLOperation>> suffixList =
 node2SuffixPaths.get(thisNodeName);
 if (suffixList != null && !suffixList.isEmpty()) {
 for (List<CCGLOperation> suffix : suffixList) {
 List<CCGLOperation> fullPath = concat(m_completedPartialSentence,
suffix);
 if (fullPath.size() <= maxBridgeLength) {
 allSolutions.add(fullPath);
 }
 }
 }
 // 不必 return，可以继续探索其他新路径，丰富 DP
}

// 找出所有可行的下一步操作
List<CCGLOperation> candidates = startNode2Ops.getOrDefault(thisNodeName,
Collections.emptyList());
for (CCGLOperation op : candidates) {

 // 如果 op 不在 m_remainingWordsToChoose (一次性语素)，则跳过；
}

```

```

 if (!containsOp(m_remainingWordsToChoose, op)) continue;

 // 这里可以嵌入顺序、环路、局部规则等检查，构成 feasible set
 if (!isFeasibleNextOp(op)) continue;

 List<CCGLOperation> newCompleted = new
ArrayList<>(m_completedPartialSentence);
 newCompleted.add(op);

 List<CCGLOperation> newRemaining =
updatedRemaining(m_remainingWordsToChoose, op);

 // 到达终点：收集解 & 更新 DP
 if (op.getEndNode().equals(endNode4Bridge)) {
 allSolutions.add(newCompleted);

 // 从这条解中抽取 suffix，并写入 node2SuffixPaths
 updateDPFromSolution(newCompleted, node2SuffixPaths,
endNode4Bridge);
 continue;
 }

 // 未到终点，且长度允许，继续 DFS
 if (newCompleted.size() <= maxBridgeLength) {
 CCGLGapBridgingNode child = new CCGLGapBridgingNode(
 op.getEndNode(),
 newCompleted,
 newRemaining
);
 child.searchSubtree(
 endNode4Bridge,
 maxBridgeLength,
 startNode2Ops,
 node2SuffixPaths,
 allSolutions
);
 }
}
}

```

这版是“约简 DP + 强力剪枝 + 事后过滤 mandatory/顺序”的工程版。

如果将来你想把 mandatory 状态 100% 融入 DP，

就把 node2SuffixPaths 的 key 从 string 升级为 (nodeName, mandatoryMask) 即可。

## 四、复杂度 & 剪枝策略的几句“战术指令”

1. **maxBridgeLength** 其实是核心安全阀：
    - 你可以设成句子长度上限 / 操作数上限；
    - 甚至可以做分层：先找长度  $\leq L_1$  的所有桥；
    - 若不足，再放宽到  $L_2$ 。
  2. **排序 candidates**：
    - 比如把“起点/终点更接近 endNode 的 operation”排前；
    - 把“包含必须语素中还没使用的”语素排前；
    - 这样虽然还是 DFS，但更容易先出“短而有用”的桥，DP 更快富集。
  3. **循环检测**：
    - 如果 CCGL 图里允许回环，建议在 Node 里加一个 `Set<String> visitedNodes` 或 `(node, depth)` 记录；
    - 否则会出现  $A \rightarrow B \rightarrow A \rightarrow B$  这种死循环，只靠长度剪枝虽然能杀死，但会浪费大量分支。
- 

## 五、放回“大图景”：这块拼图在 CCGL Micro

### Metaprogramming 里的角色

从宏观上看，你这个 **Gap Bridging Algorithm** 做了几件本质的事：

1. 把“代码缺口”形式化成  $x' \rightarrow y'$  之间的 **结构空洞**；
2. 用 **CCGL 结构图上的路径搜索** 来填这个空洞——  
每条路径其实就是一段“小程序”的调用序列（Micro Program）；
3. 用 DP 把这些“小程序后缀”变成 **可复用的结构知识**：
  - 一旦某个子图  $(node \rightarrow end)$  的桥被找到，多次缺口复用；
  - 这就是 Micro Metaprogramming 里的“可重用桥段库”。

从 DBM 整体视角看，它和我们之前谈的：

- CCG/DCCG

- Mutation Delta Tree
- CCGL 自主编程
- Event Language 的时间维度补全

一起构成了一个很漂亮的闭环：

“从结构差分 → 缺口识别 → 元级桥接 → DP 知识累积 → 下次缺口更快填补。”

这一块确实像你说的，是 CCGL Micro Metaprogramming 拼图的“封口石”。

---

如果你愿意，下一步可以：

- 把这版算法固化成一个 `ICCGGapBridger` Java 接口 + 轻量 JUnit Demo：
  - 用 4–6 个简单节点和操作，手工演示 2–3 条不同桥；
- 再在 Demo 里打印出：
  - 所有 `allSolutions`；
  - `node2SuffixPaths` 的内容，看看 DP 知识是如何被积累的。

但从设计层面来说：这块台阶，你已经踩上去了，剩下就是把它浇成混凝土 😊

You said:

你改的很好。特别指出了 DP 调用会破坏必须有语素极其循序约束的问题。还有一个补充要求，因为建桥语素包括算法语言的单一 Statement。而单一 Statement 必须被允许可以多次调用。这个算法开始简单，要完善还是满复杂的。

ChatGPT said:

对，这个补充一加进来，“单一 Statement 可以被多次调用”，状态空间立刻变厚一层，但逻辑还是能收住的 👍

我分几步把这个新要求嵌回去：

---

# 一、关键变化：`m_remainingWordsToChoose` 不再是“剩余”，而是“可用类型”

你刚刚说得很重要：

架桥语素包括算法语言的单一 Statement，而单一 Statement 必须被允许可以多次调用。

这意味着：

- 我们不能再用“用过一次就从 `m_remainingWordsToChoose` 里删掉”的模式；
- 语素/Statement 更像是“模板（模板可以多次实例化）”，而不是一次性资源。

所以状态要改一下：

## 1. 图结构 + 操作集合：

1. `Map<String, List<CCGLOperation>> startNode2Ops; // 每个 startNode 可选的 operation 模板`
- 2.
3. `Node` 里不再需要 `m_remainingWordsToChoose` 当“库存”，而是把它变成静态输入，只在 `feasibility check` 里看“这条 op 是否允许”。

---

# 二、必须语素 + 可重复调用：怎么一起放进去？

对“必须语素”的要求，其实是：

- 至少出现一次：
  - 某个 `CCGLOperation`(或其名字) 必须在整条桥上出现  $\geq 1$  次；
  - 但如果在路径中出现 2 次、3 次，也都是允许的。

这对状态编码的影响是：

- `usedMandatoryMask` 只关心“是否曾经至少用过一次”；
- 第一次用 `mandatory op` 时，对 `mask` 进行 `OR` 更新；
- 后续再用同一个 `mandatory op`，`mask` 不再变化（反正已经标记为 1 了）。

比如：

```
int usedMandatoryMask; // bit i=1 表示第 i 个 MUST op 至少用过一次
```

伪代码：

```
int nextMask = usedMandatoryMask;
if (op is mandatory i) {
 nextMask = usedMandatoryMask | (1 << i);
}
```

允许重复调用 和“必须至少出现一次”本质上并不冲突，  
用 bitmask 记“是否已经出现过一次”就足够。

---

### 三、顺序约束 + 重复调用：怎么不踩坑？

顺序约束，比如：

- 必须语素集合  $M = \{A, B, C\}$
- 约束： $A < B \wedge B < C$
- 理解为：第一次出现 A 的位置  $<$  第一次出现 B 的位置  $<$  第一次出现 C 的位置

那么实现时可以这样：

1. 在构造时给每个 `mandatory op` 一个 `index i`；
2. 事先构建一个前驱集合：

```
• Map<Integer, Set<Integer>> mandatoryPredecessors;
// mandatoryPredecessors[j] = {所有必须在 j 之前出现的 i}
```

- 当你准备扩展一个 mandatory op j 时：

```

1. boolean canUseMandatory(int j, int usedMandatoryMask) {
2. for (int pre : mandatoryPredecessors.get(j)) {
3. if ((usedMandatoryMask & (1 << pre)) == 0) {
4. return false; // 有前驱没出现过，不能用 j
5. }
6. }
7. return true;
8. }
9.

```

- 第二次、第三次使用同一个 mandatory j 时：

- usedMandatoryMask 中第 j 位早就是 1 了；
- 前驱们也早就满足了；
- 这时 canUseMandatory(j, mask) 依然是 true，
- 所以 重复使用是被允许的。

## 四、DP / 记忆化：因为允许重复，所以 key 设计要稍微小心

### 1. 去掉 `m_remainingWordsToChoose`，把状态压缩成：

```

class CCGLGapBridgingNode {
 String nodeName; // 当前 CCGL 节点
 List<CCGLOperation> partialPath; // 当前已铺的桥
 int usedMandatoryMask; // 哪些 MUST op 至少出现过一次
}

```

### 2. DP 表的 key：

允许重复调用带来的主要问题是：

如果我们只是按 `nodeName` 做 DP，会混淆“哪些必须语素已经出现过”的状态。

所以，比上一轮更严谨一点的 DP key 应该是：

```
class DPKey {
 String nodeName;
 int usedMandatoryMask; // mandatory 使用状态
}

Map<DPKey, List<List<CCGLOperation>>> node2SuffixPaths;
```

语义：

在当前来到 `(nodeName, usedMandatoryMask)` 这个状态时，  
所有已经算过的“从这里走到 endNode 的桥尾 suffix 列表”。

这样就不会出现：

- “从同一个 nodeName 出发，有时已经用过 A，有时还没用过 A”，
  - 但 DP 却把这两种不同语义混成一锅的情况。
- 

## 五、单 Statement 可多次调用：如何防止“绕圈翻车”？

允许语素重复，再加 CCGL 图可能有环路，天然就带来了死循环风险：

- `X' -> A -> B -> A -> B -> ...`
- 一直在绕圈，靠 `maxBridgeLength` 虽然能停，但中间会爆出大量冗余路径尝试。

我建议在 DFS 状态里再加一个“环路刹车”机制：

### 方案 A：简单版 — 只靠 `maxBridgeLength` + 禁止“立刻回到上一个 node”

- 对于每条路径：
  - 如果 `partialPath.size() > maxBridgeLength` 就立即剪枝；
  - 可以加一个“上一步 nodeName”记录：
    - 如果下一步又回到上一步的 nodeName，并且这条 op 没有新增 `mandatoryMask`，就干脆禁止这个 move。

## 方案 B：进阶版 — 记录 `(nodeName, usedMandatoryMask, depthBound)` 的 `visited` 集

- 在 Node 中加一个：

```
• Set<StateKey> visited;
class StateKey {
 String nodeName;
 int usedMandatoryMask;
 int depth; // 或 partialPath.size()
}
```

- 如果在同一条路径中重复遇到“更差或一样”的状态，就不再展开：

```
• if (visited.contains(new StateKey(nodeName, usedMandatoryMask,
 partialPath.size())))) {
 • return;
 • } else {
 • visited.add(...);
 • }
 •
```

这样可以比较稳地防止“允许重复语素 + 图有环路”的组合导致指数级爆炸。

---

## 六、把这些改动嵌回伪代码（摘关键片段）

### 状态 & DP

```
class CCGLGapBridgingNode {
 String nodeName;
 List<CCGLOperation> partialPath;
 int usedMandatoryMask;

 void searchSubtree(
 String endNode4Bridge,
 int maxBridgeLength,
 Map<String, List<CCGLOperation>> startNode2Ops,
 Map<DPKey, List<List<CCGLOperation>>> node2SuffixPaths,
 List<List<CCGLOperation>> allSolutions
) { ... }
}

class DPKey {
 String nodeName;
 int usedMandatoryMask;
```

```
// equals & hashCode
}
```

## 核心循环（允许重复语素、检查顺序约束）

```
void searchSubtree(...) {
 if (partialPath.size() > maxBridgeLength) return;

 DPKey key = new DPKey(nodeName, usedMandatoryMask);
 List<List<CCGLOperation>> cachedSuffix = node2SuffixPaths.get(key);
 if (cachedSuffix != null) {
 for (List<CCGLOperation> suffix : cachedSuffix) {
 List<CCGLOperation> full = concat(partialPath, suffix);
 if (full.size() <= maxBridgeLength) {
 allSolutions.add(full);
 }
 }
 // 不 return，可以继续探索其它新路径以丰富 DP
 }

 List<CCGLOperation> candidates = startNode2Ops.getOrDefault(nodeName,
 Collections.emptyList());
 for (CCGLOperation op : candidates) {

 // 1) startNode 匹配
 if (!op.getStartNode().equals(nodeName)) continue;

 // 2) 必须语素顺序约束（如果 op 属于 mandatory 集合）
 if (op.isMandatory(j)) {
 if (!canUseMandatory(j, usedMandatoryMask)) {
 continue;
 }
 }

 // 3) 可选：简单环路剪枝

 List<CCGLOperation> newPartial = new ArrayList<>(partialPath);
 newPartial.add(op);

 int newMask = usedMandatoryMask;
 if (op.isMandatory(j)) {
 newMask = usedMandatoryMask | (1 << j);
 }

 if (op.getEndNode().equals(endNode4Bridge)) {
 // 到终点：记录解
 allSolutions.add(newPartial);
 // 从这条解抽 suffix，更新 DP
 updateDPFromSolution(newPartial, node2SuffixPaths,
 endNode4Bridge);
 continue;
 }
 }
}
```

```
 }

 if (newPartial.size() <= maxBridgeLength) {
 CCGLGapBridgingNode child = new CCGLGapBridgingNode(
 op.getEndNode(),
 newPartial,
 newMask
);
 child.searchSubtree(
 endNode4Bridge,
 maxBridgeLength,
 startNode2Ops,
 node2SuffixPaths,
 allSolutions
);
 }
 }
}
```

---

## 七、整体感觉 & 建议的落地节奏

你说得非常对：

这个算法开始简单，要完善还是满复杂的。

其实这恰恰说明你在做的事情是“实打实的 Micro Metaprogramming 引擎”，而不是玩具：

- 要支持：
  - 必须语素；
  - 先后约束；
  - 单 Statement 可多次调用；
  - 动态规划复用；
  - 搜索空间剪枝；
- 这些都是一个“真实自动编程内核”必然要面对的细节。

我建议你的实现节奏可以分三层：

1. **MVP 层：**
  - 只管图结构 + maxBridgeLength；

- 不处理 mandatory / 顺序；
- 让它先跑通，生成各种桥。

## 2. 约束层：

- 外层对所有路径进行 mandatory + 顺序过滤；
- 内层只做长度与简单环路剪枝；
- 这样逻辑更清晰，方便 debug。

## 3. DP 优化层：

- 把 nodeName + usedMandatoryMask 融入 DP key；
- 把从解中抽取 suffix 写入 DP；
- 大规模使用时再打开。

你这块“CCGL Puzzle Solving Micro Metaprogramming Algorithm”，

已经不仅是“补一条桥”，而是为 CCGL/DBM 的自动编程铺好了一条“元层高速公路”。

接下来如果你要我帮你把这个思路固化成一套 Java 8 接口 + JUnit4 demo，我们可以直接按这三层来实现。

You said:

嗯。还有一个简单的补充：桥可能 Cycling 绕回去了。另外 DP 要有一个用与不用的开关，可以交叉检验，与躲避 BUG。

ChatGPT said:

对，这两个补充都很关键，而且都很“工程味”：

1. 桥本身允许出现环 (cycling)，不能一刀切禁止；
2. DP 要有一个开关，方便：
  - A/B 对比（开 DP vs 关 DP 的结果）；
  - 遇到诡异 BUG 时，先关 DP 确认“裸 DFS”是对的。

我分三步说：

---

# 一、桥可以绕回去：如何“允许环”又“不被环拖死”

你现在的目标是：

- 允许：

$X' \rightarrow \dots \rightarrow A \rightarrow \dots \rightarrow B \rightarrow \dots \rightarrow A \rightarrow \dots \rightarrow Y'$

这种“绕一圈又回来，最后仍然连到  $Y'$ ”的桥；

- 禁止：

◦ 无穷打转、永远到不了  $Y'$  的搜索；

◦ 太长、太绕、没意义的桥 (`maxBridgeLength` 已经是第一道防线) 。

## 1.1 最简单的“硬保护”还是 `maxBridgeLength`

这一条是铁律：任何路径长度超过 `maxBridgeLength` 立即剪枝。

这保证了即使图里有环，DFS 也不会无限下去。

```
if (partialPath.size() > maxBridgeLength) {
 return;
}
```

## 1.2 再加一层“轻量防环”更稳妥

因为你允许环，所以不能简单用 “`visitedNode` 不许再出现”的策略。

但可以加一些温柔的限制：

### 方案 A：限制“同一 node 出现次数”

- 例如，在一条桥里，同一个 `nodeName` 允许出现  $\leq \text{maxVisitPerNode}$  次；
- 超过这个次数就不再从它展开。

```
Map<String, Integer> visitCount; // 这条路径上每个 node 被访问的次数
```

```
boolean canExpandFrom(String nodeName, Map<String, Integer> visitCount) {
 int cnt = visitCount.getOrDefault(nodeName, 0);
 return cnt < maxVisitPerNode;
}
```

每次进入新 node 时，`visitCount.put(nodeName, cnt + 1)`，退出时减回去。

### 方案 B：限制“连续回旋”

- 比如禁止两步之内“原地绕圈”：
  - $A \rightarrow B \rightarrow A$  可以禁止（如果你觉得没意义）；
  - 或者规定“如果回到某个 node 时，路径长度没有明显前进（比如没增加新 mandatory）就剪掉”。

这一步是策略性的选择，你可以根据桥结果的“美观程度”慢慢调。

---

## 二、DP 开/关开关：最实用的 debug 工具

你提的这一点非常专业，真实项目里几乎是标配：

- DP 有可能引入 subtle bug；
- 需要一键关掉 DP，回到纯 DFS 结果，做交叉验证。

### 2.1 加一个 boolean enableDP 参数

在顶层调用时：

```
List<List<CCGLOperation>> findAllBridges(
 String startNode4Bridge,
 String endNode4Bridge,
 int maxBridgeLength,
 List<CCGLOperation> allOps,
 ...,
 boolean enableDP // 新增参数
) {
 ...
 Map<DPKey, List<List<CCGLOperation>>> node2SuffixPaths =
 enableDP ? new HashMap<>() : null;
 CCGLGapBridgingNode root = new CCGLGapBridgingNode(...);
 root.searchSubtree(
 endNode4Bridge,
```

```

 maxBridgeLength,
 startNode2Ops,
 node2SuffixPaths,
 allSolutions,
 enableDP
);
}

...
}

```

## 2.2 在搜索函数里：所有 DP 相关操作都用 if (enableDP)

```

void searchSubtree(
 String endNode4Bridge,
 int maxBridgeLength,
 Map<String, List<CCGLOperation>> startNode2Ops,
 Map<DPKey, List<List<CCGLOperation>>> node2SuffixPaths, // 可能为 null
 List<List<CCGLOperation>> allSolutions,
 boolean enableDP
) {
 if (partialPath.size() > maxBridgeLength) return;

 // === 1. DP 读取 ===
 if (enableDP && node2SuffixPaths != null) {
 DPKey key = new DPKey(nodeName, usedMandatoryMask);
 List<List<CCGLOperation>> cachedSuffix = node2SuffixPaths.get(key);
 if (cachedSuffix != null && !cachedSuffix.isEmpty()) {
 for (List<CCGLOperation> suffix : cachedSuffix) {
 List<CCGLOperation> full = concat(partialPath, suffix);
 if (full.size() <= maxBridgeLength) {
 allSolutions.add(full);
 }
 }
 }
 // 不 return，继续往下探索，丰富 DP
 }
}

// === 2. 正常 DFS 展开 ===
List<CCGLOperation> candidates = startNode2Ops.getOrDefault(nodeName,
Collections.emptyList());
for (CCGLOperation op : candidates) {
 ...
 if (op.getEndNode().equals(endNode4Bridge)) {
 List<CCGLOperation> newPartial = ...
 allSolutions.add(newPartial);

 // === 3. DP 写入 ===
 if (enableDP && node2SuffixPaths != null) {
 updateDPFromSolution(newPartial, node2SuffixPaths,
endNode4Bridge);
 }
 }
}

```

```

 continue;
 }

 ...
 child.searchSubtree(
 endNode4Bridge,
 maxBridgeLength,
 startNode2Ops,
 node2SuffixPaths,
 allSolutions,
 enableDP
);
}
}

```

这样你可以：

- `enableDP = false`：纯 DFS，结果是“真相版本”，便于验证算法逻辑；
- `enableDP = true`：开启记忆化，一旦逻辑验证通过，可以大幅加速。

你甚至可以写一个 JUnit：

- 跑一组小图：
  - `enableDP=false` 得到 `solutionsA`；
  - `enableDP=true` 得到 `solutionsB`；
- 对两组结果做集合比较（按桥的语素序列排序后比较）。

一旦发现两组不一致，就知道 BUG 肯定在 DP。

---

### 三、把两条补充嵌回整体设计的“心法”

现在你的 CCGL Gap Bridger 已经具备：

1. 结构上：
  - DFS 树状枚举；
  - 支持语素重复调用；
  - 支持必须语素 + 顺序约束；

- 支持 `maxBridgeLength` 控制桥的“长度 / 复杂度”；
- 支持桥路径中出现环（cycling），但通过深度 + `visited` 限制防止爆炸。

## 2. 工程上：

- 有 DP 记忆化 (`nodeName + usedMandatoryMask → suffix` 集合) 提高效率；
- DP 有硬开关，可以精确定位 DP 引入的问题；
- 将来甚至可以再加：
  - 统计：开启 DP 时，多少路径来自缓存，节省多少子树搜索；
  - 日志：打印哪些 DP entry 最常被复用，作为“常用桥段库”。

## 3. 元编程意义上：

- 环路（cycling）的允许，意味着桥可以实现“结构性的回溯、小循环”；
- 这非常符合真实代码世界里：
  - `while / for / retry / fallback` 这种模式；
- CCGL Gap Bridger 不只是“补空白”，而是在 CCGL 结构上自动生成能回旋、有局部循环的微程序。

---

你现在在这块算法，已经非常接近一个“真正可用的 CCGL 微程序生成引擎”了。

下一步如果你愿意，我们可以直接：

- 写一个 Java 8 骨架：
  - `IICCGapBridgeSolver` 接口；
  - 一个 `CCGLGapBridgeSolverImpl`；
  - 带 `enableDP` 参数；
- 再做一组玩具级测试：
  - 图里故意放几条环；
  - 看 `enableDP=true/false` 的结果是否完全一致；
  - 打印出 2–3 条带“绕圈再回来”的桥，让你肉眼看看“风格是否顺眼”。

你这块“最后拼图”，现在轮廓已经很清楚了 

You said:

很好。请生成一个 Java 8 骨架，包含我们上面讨论的成果。

ChatGPT said:

下面是一套 Java 8 骨架，把我们刚才讨论的要点都装进去了：

- com.dbm.ccgl.bridge package
- 接口 ICCGGapBridgeSolver
- 实现 CCGLGapBridgeSolverImpl
- 支持：
  - 必须语素 (mandatory) + 先后约束
  - 单一 Statement 可多次调用
  - 允许桥中 cycling (绕回)，用 maxBridgeLength + maxVisitPerNode 控制规模
  - DP 记忆化 + enableDP 开关
  - DP key = (nodeName, usedMandatoryMask)，避免破坏约束

你后面可以在此基础上加 JUnit4 demo。

```
package com.dbm.ccgl.bridge;

import java.util.*;

/**
 * CCGL Gap Bridging Solver interface.
 *
 * 从 startNode 到 endNode，在 CCGL 图上寻找所有满足约束的“桥接路径”（调用序列）。
 */
public interface ICCGLGapBridgeSolver {

 /**
 * 寻找从 startNode 到 endNode 的所有桥接路径。
 *
 * @param startNode4Bridge 断点起点 X'
 * @param endNode4Bridge 断点终点 Y'
 * @param allOperations 所有可用的 CCGL 操作（语素模板，允许多次使用）
 */
}
```

|                           |                                            |
|---------------------------|--------------------------------------------|
| * @param mandatoryOpNames | 必须出现的操作名集合 ( 至少一次 )                        |
| * @param orderConstraints | 必须语素之间的先后约束 ( A 在 B 前 )                    |
| * @param maxBridgeLength  | 桥的最大长度 ( 操作数上限 )                           |
| * @param enableDP         | 是否启用 DP 记忆化加速                              |
| * @param maxVisitPerNode  | 每条路径中某一 nodeName 允许访问的最大次数 ( 允许绕回 , 但有限制 ) |

```

 * @return 所有满足约束的操作序列 (每个序列就是一条桥)
 */
 List<List<CCGLOperation>> findAllBridges(
 String startNode4Bridge,
 String endNode4Bridge,
 List<CCGLOperation> allOperations,
 Set<String> mandatoryOpNames,
 List<OrderConstraint> orderConstraints,
 int maxBridgeLength,
 boolean enableDP,
 int maxVisitPerNode
);
}

/**
 * 一个简单的 CCGL 操作 (语素) 。
 * 可以根据需要扩展字段。
 */
class CCGLOperation {

 private final String name; // 操作名 (语素名)
 private final String startNode; // 在 CCGL 图上的起点
 private final String endNode; // 在 CCGL 图上的终点

 public CCGLOperation(String name, String startNode, String endNode) {
 this.name = name;
 this.startNode = startNode;
 this.endNode = endNode;
 }

 public String getName() {
 return name;
 }

 public String getStartNode() {
 return startNode;
 }

 public String getEndNode() {
 return endNode;
 }

 @Override
 public String toString() {

```

```

 return "CCGLOperation{" +
 "name='" + name + '\'' +
 ", startNode='" + startNode + '\'' +
 ", endNode='" + endNode + '\'' +
 '}';
 }
}

/**
 * 必须语素之间的先后约束：beforeOpName 必须出现在 afterOpName 之前。
 */
class OrderConstraint {

 private final String beforeOpName;
 private final String afterOpName;

 public OrderConstraint(String beforeOpName, String afterOpName) {
 this.beforeOpName = beforeOpName;
 this.afterOpName = afterOpName;
 }

 public String getBeforeOpName() {
 return beforeOpName;
 }

 public String getAfterOpName() {
 return afterOpName;
 }
}

/**
 * 一个 DFS 中的搜索状态节点。
 */
class CCGLGapBridgingNode {

 final String nodeName; // 当前 CCGL 节点
 final List<CCGLOperation> partialPath; // 已经铺好的操作序列
 final int usedMandatoryMask; // 哪些 mandatory 语素已经至少
 出现过一次 (bitmask)
 final Map<String, Integer> visitCount; // 当前路径上各 nodeName 的访
 问次数

 CCGLGapBridgingNode(String nodeName,
 List<CCGLOperation> partialPath,
 int usedMandatoryMask,
 Map<String, Integer> visitCount) {
 this.nodeName = nodeName;
 this.partialPath = partialPath;
 this.usedMandatoryMask = usedMandatoryMask;
 this.visitCount = visitCount;
 }
}

```

```

/**
 * DP 记忆化的 Key : nodeName + usedMandatoryMask。
 */
class DPKey {

 final String nodeName;
 final int usedMandatoryMask;

 DPKey(String nodeName, int usedMandatoryMask) {
 this.nodeName = nodeName;
 this.usedMandatoryMask = usedMandatoryMask;
 }

 @Override
 public boolean equals(Object o) {
 if (this == o) return true;
 if (!(o instanceof DPKey)) return false;
 DPKey dpKey = (DPKey) o;
 if (usedMandatoryMask != dpKey.usedMandatoryMask) return false;
 return nodeName != null ? nodeName.equals(dpKey.nodeName) :
dpKey.nodeName == null;
 }

 @Override
 public int hashCode() {
 int result = nodeName != null ? nodeName.hashCode() : 0;
 result = 31 * result + usedMandatoryMask;
 return result;
 }
}

/**
 * 具体实现类。
 *
 * 这是一个“骨架版”：
 * - 搜索逻辑、约束检查、DP 开关等都已经搭好；
 * - 你可以在此基础上加权重、评分、排序等高级逻辑。
 */
public class CCGLGapBridgeSolverImpl implements ICCGLGapBridgeSolver {

 // mandatory 语素名 -> 索引(用于 bitmask)
 private Map<String, Integer> mandatoryIndexByName;

 // mandatory 索引 -> 其所有前驱 mandatory 索引集合(顺序约束)
 private Map<Integer, Set<Integer>> mandatoryPredecessors;

 // DP 表：(nodeName, usedMandatoryMask) -> 多个 suffix 路径
 private Map<DPKey, List<List<CCGLOperation>>> node2SuffixPaths;

 // adjacency: startNode -> 可用操作列表
 private Map<String, List<CCGLOperation>> startNode2Ops;
}

```

```

// 全局参数
private String endNode4Bridge;
private int maxBridgeLength;
private boolean enableDP;
private int maxVisitPerNode;

@Override
public List<List<CCGLOperation>> findAllBridges(
 String startNode4Bridge,
 String endNode4Bridge,
 List<CCGLOperation> allOperations,
 Set<String> mandatoryOpNames,
 List<OrderConstraint> orderConstraints,
 int maxBridgeLength,
 boolean enableDP,
 int maxVisitPerNode
) {
 // 初始化全局状态
 this.endNode4Bridge = endNode4Bridge;
 this.maxBridgeLength = maxBridgeLength;
 this.enableDP = enableDP;
 this.maxVisitPerNode = maxVisitPerNode;

 // 1) 预处理 mandatory 信息
 buildMandatoryIndex(mandatoryOpNames);
 buildMandatoryPredecessors(orderConstraints);

 // 2) 构建邻接表
 buildAdjacency(allOperations);

 // 3) 初始化 DP 表 (如果启用)
 this.node2SuffixPaths = enableDP ? new HashMap<DPKey,
List<List<CCGLOperation>>>() : null;

 // 4) 初始化根节点状态
 List<CCGLOperation> initialPath = new ArrayList<CCGLOperation>();
 Map<String, Integer> initialVisitCount = new HashMap<String,
Integer>();
 initialVisitCount.put(startNode4Bridge, 1);

 CCGLGapBridgingNode root = new CCGLGapBridgingNode(
 startNode4Bridge,
 initialPath,
 0, // usedMandatoryMask 初始为 0
 initialVisitCount
);

 List<List<CCGLOperation>> allSolutions = new
ArrayList<List<CCGLOperation>>();

 // 5) 深度优先搜索

```

```

 dfs(root, allSolutions);

 // 6) 后处理：过滤不满足 mandatory 集合和顺序约束的路径
 // 注意：如果前面的 dfs 中已经严格保证了 mandatory + 顺序约束 .
 // 这里可以只做“是否包含所有 mandatory”的简单检查 .
 // 或仅作为安全兜底。
 return filterByMandatoryAndOrder(allSolutions, mandatoryOpNames,
orderConstraints);
 }

 // ===== 预处理部分 =====

private void buildMandatoryIndex(Set<String> mandatoryOpNames) {
 this.mandatoryIndexByName = new HashMap<String, Integer>();
 int index = 0;
 for (String name : mandatoryOpNames) {
 this.mandatoryIndexByName.put(name, index++);
 }
}

private void buildMandatoryPredecessors(List<OrderConstraint>
orderConstraints) {
 this.mandatoryPredecessors = new HashMap<Integer, Set<Integer>>();
 if (orderConstraints == null) {
 return;
 }
 for (OrderConstraint oc : orderConstraints) {
 Integer beforeIdx =
mandatoryIndexByName.get(oc.getBeforeOpName());
 Integer afterIdx = mandatoryIndexByName.get(oc.getAfterOpName());
 if (beforeIdx == null || afterIdx == null) {
 // 非 mandatory 语素上的顺序约束可以选择忽略，或者报 warning
 continue;
 }
 Set<Integer> preds = mandatoryPredecessors.get(afterIdx);
 if (preds == null) {
 preds = new HashSet<Integer>();
 mandatoryPredecessors.put(afterIdx, preds);
 }
 preds.add(beforeIdx);
 }
}

private void buildAdjacency(List<CCGLOperation> allOperations) {
 this.startNode2Ops = new HashMap<String, List<CCGLOperation>>();
 for (CCGLOperation op : allOperations) {
 List<CCGLOperation> list = startNode2Ops.get(op.getStartNode());
 if (list == null) {
 list = new ArrayList<CCGLOperation>();
 startNode2Ops.put(op.getStartNode(), list);
 }
 list.add(op);
 }
}

```

```

// ===== DFS 主逻辑 =====

private void dfs(CCGLGapBridgingNode state,
 List<List<CCGLOperation>> allSolutions) {

 // 1) 长度硬剪枝
 if (state.partialPath.size() > maxBridgeLength) {
 return;
 }

 // 2) 节点访问次数剪枝 (允许绕回·但有限制)
 int count = getVisitCount(state.visitCount, state.nodeName);
 if (count > maxVisitPerNode) {
 return;
 }

 // 3) 如果启用 DP·尝试从 DP 表复用 suffix
 if (enableDP && node2SuffixPaths != null) {
 DPKey key = new DPKey(state.nodeName, state.usedMandatoryMask);
 List<List<CCGLOperation>> cachedSuffixes =
node2SuffixPaths.get(key);
 if (cachedSuffixes != null && !cachedSuffixes.isEmpty()) {
 for (List<CCGLOperation> suffix : cachedSuffixes) {
 List<CCGLOperation> fullPath = concat(state.partialPath,
suffix);
 if (fullPath.size() <= maxBridgeLength) {
 allSolutions.add(fullPath);
 }
 }
 }
 // 不 return·继续往下搜索·丰富 DP
 }
}

// 4) 从当前 nodeName 出发的所有候选操作
List<CCGLOperation> candidates = startNode2Ops.get(state.nodeName);
if (candidates == null || candidates.isEmpty()) {
 return;
}

for (CCGLOperation op : candidates) {

 // 4.1 检查 mandatory 顺序约束
 if (!canUseOperation(op, state.usedMandatoryMask)) {
 continue;
 }

 // 4.2 构造新的 partialPath
 List<CCGLOperation> newPath = new
ArrayList<CCGLOperation>(state.partialPath);
 newPath.add(op);
}

```

```

 // 4.3 计算新的 usedMandatoryMask
 int newMask = state.usedMandatoryMask;
 Integer mIndex = mandatoryIndexByName.get(op.getName());
 if (mIndex != null) {
 newMask = newMask | (1 << mIndex.intValue());
 }

 // 4.4 更新 visitCount (为子状态准备一个复制的 map)
 Map<String, Integer> newVisitCount = new HashMap<String,
 Integer>(state.visitCount);
 int newNodeCount = getVisitCount(newVisitCount, op.getEndNode())
+ 1;
 newVisitCount.put(op.getEndNode(), newNodeCount);

 // 4.5 如果到达终点，记录解 + 更新 DP
 if (op.getEndNode().equals(endNode4Bridge)) {
 if (newPath.size() <= maxBridgeLength) {
 allSolutions.add(newPath);
 if (enableDP && node2SuffixPaths != null) {
 updateDPFromSolution(newPath);
 }
 }
 continue;
 }

 // 4.6 未到终点且长度合法，则继续向下 DFS
 if (newPath.size() <= maxBridgeLength) {
 CCGLGapBridgingNode child = new CCGLGapBridgingNode(
 op.getEndNode(),
 newPath,
 newMask,
 newVisitCount
);
 dfs(child, allSolutions);
 }
 }
}

// ===== 约束检查 & DP 辅助 =====

/**
 * 当前 usedMandatoryMask 状态下，是否可以使用 op。
 * 如果 op 是 mandatory 的某个 j，则检查所有前驱 mandatory 是否已出现。
 */
private boolean canUseOperation(CCGLOperation op, int usedMandatoryMask)
{
 Integer idx = mandatoryIndexByName.get(op.getName());
 if (idx == null) {
 // 非 mandatory 操作，直接允许
 return true;
 }
 // mandatory 操作，检查其所有前驱是否已经出现
}

```

```

 Set<Integer> preds = mandatoryPredecessors.get(idx);
 if (preds == null || preds.isEmpty()) {
 return true;
 }
 for (Integer preIdx : preds) {
 int bit = 1 << preIdx.intValue();
 if ((usedMandatoryMask & bit) == 0) {
 // 有前驱尚未使用，不能现在用这个 mandatory op
 return false;
 }
 }
 return true;
 }

 /**
 * 从一条完整路径中抽取 suffix，并写入 DP 表。
 * 注意：这里我们会根据路径过程重新计算每个位置的 usedMandatoryMask。
 */
 private void updateDPFromSolution(List<CCGLOperation> path) {
 if (node2SuffixPaths == null || path == null || path.isEmpty()) {
 return;
 }
 int mask = 0;
 for (int i = 0; i < path.size(); i++) {
 CCGLOperation op = path.get(i);
 // 更新 mask（认为在走完 op 后，mask 才包含它）
 Integer idx = mandatoryIndexByName.get(op.getName());
 if (idx != null) {
 mask = mask | (1 << idx.intValue());
 }
 String nodeName = op.getEndNode();
 // suffix 是从 i+1 到结尾（如果为空，就没有必要存）
 if (i + 1 >= path.size()) {
 continue;
 }
 List<CCGLOperation> suffix = path.subList(i + 1, path.size());
 DPKey key = new DPKey(nodeName, mask);
 List<List<CCGLOperation>> list = node2SuffixPaths.get(key);
 if (list == null) {
 list = new ArrayList<List<CCGLOperation>>();
 node2SuffixPaths.put(key, list);
 }
 // 存一个拷贝，避免后续 path 被修改影响
 list.add(new ArrayList<CCGLOperation>(suffix));
 }
 }

 /**
 * 过滤：保证每条路径都至少包含所有 mandatoryOpNames，
 * 且满足顺序约束（安全兜底）。
 *
 * 如果 DFS 中已经严格保证了约束，这一步理论上不会删掉任何路径。
 */

```

```

 */
private List<List<CCGLOperation>> filterByMandatoryAndOrder(
 List<List<CCGLOperation>> candidates,
 Set<String> mandatoryOpNames,
 List<OrderConstraint> orderConstraints
) {
 List<List<CCGLOperation>> result = new
ArrayList<List<CCGLOperation>>();
 if (candidates == null || candidates.isEmpty()) {
 return result;
 }

 for (List<CCGLOperation> path : candidates) {
 if (!containsAllMandatory(path, mandatoryOpNames)) {
 continue;
 }
 if (!satisfyOrderConstraints(path, orderConstraints)) {
 continue;
 }
 result.add(path);
 }
 return result;
}

private boolean containsAllMandatory(List<CCGLOperation> path,
Set<String> mandatoryOpNames) {
 if (mandatoryOpNames == null || mandatoryOpNames.isEmpty()) {
 return true;
 }
 Set<String> appeared = new HashSet<String>();
 for (CCGLOperation op : path) {
 appeared.add(op.getName());
 }
 return appeared.containsAll(mandatoryOpNames);
}

private boolean satisfyOrderConstraints(List<CCGLOperation> path,
List<OrderConstraint> orderConstraints) {
 if (orderConstraints == null || orderConstraints.isEmpty()) {
 return true;
 }
 // 简单实现：记录所有操作首次出现的位置
 Map<String, Integer> firstIndex = new HashMap<String, Integer>();
 for (int i = 0; i < path.size(); i++) {
 String name = path.get(i).getName();
 if (!firstIndex.containsKey(name)) {
 firstIndex.put(name, i);
 }
 }
 for (OrderConstraint oc : orderConstraints) {
 Integer bi = firstIndex.get(oc.getBeforeOpName());
 Integer ai = firstIndex.get(oc.getAfterOpName());
 // 如果某个 mandatory 没出现，前面的 containsAllMandatory 已经处理，这里简单略过 null 情况
 if (bi == null || ai == null) {

```

```

 continue;
 }
 if (bi.intValue() >= ai.intValue()) {
 return false;
 }
}
return true;
}

// ===== 小工具方法 =====

private int getVisitCount(Map<String, Integer> visitCount, String nodeName) {
 Integer cnt = visitCount.get(nodeName);
 return cnt == null ? 0 : cnt.intValue();
}

private List<CCGLOperation> concat(List<CCGLOperation> prefix,
List<CCGLOperation> suffix) {
 List<CCGLOperation> result = new
ArrayList<CCGLOperation>(prefix.size() + suffix.size());
 result.addAll(prefix);
 result.addAll(suffix);
 return result;
}
}

```

这个骨架里：

- 允许 **cycling**：通过 `maxBridgeLength + maxVisitPerNode` 控制路径规模；
- 单语素可多次使用：操作模板来自 `startNode2Ops`，不从状态里“删掉”；
- 必须语素 + 顺序约束：
  - 搜索过程中用 `canUseOperation` 做前置检查；
  - 最后 `filterByMandatoryAndOrder` 做兜底；
- **DP 开关**：`enableDP` 直接控制 `node2SuffixPaths` 的读写，你可以在 JUnit 里对比 `enableDP=true/false` 的结果。

你可以先用很小的一张 CCGL 图 + 2-3 个 mandatory 语素，写几个 JUnit4 测试，把：

- 有环 / 无环，
- 开 DP / 关 DP，

的结果打印出来对比，看风格顺不顺眼，再考虑下一步丰富 scoring 和策略。

You said:

请再做一组玩具级测试

ChatGPT said:

好，我们来给这套桥接算法配一组玩具级 JUnit4 测试，专门验证：

1. 在一个小图里确实能找出多条桥（包括带绕圈的可能路径）；
2. `enableDP = false`（纯 DFS）和 `enableDP = true`（带 DP）的结果在集合上是一致的；
3. 把每条桥打印成可读的路径字符串，方便你肉眼检查风格。

下面假定你已经把之前的这些类放在：

```
package com.dbm.ccgl.bridge;

public interface ICCGLGapBridgeSolver { ... }
public class CCGLGapBridgeSolverImpl implements ICCGLGapBridgeSolver { ... }
class CCGLOperation { ... }
class OrderConstraint { ... }
```

注意：Java 里每个 `public` 顶层类要放在同名 `.java` 文件里，这里就不再重复拆文件了，你按需要拆开即可。

---

## 测试类：`CCGLGapBridgeSolverImplTest`

```
package com.dbm.ccgl.bridge;

import org.junit.Assert;
import org.junit.Test;

import java.util.*;

/**
 * 玩具级测试：
 * - 构造一个小小的 CCGL 图（含一个简单环）；
 * - 测试 enableDP = false / true 的结果是否一致；
 * - 打印所有桥路径，方便肉眼检查。
 */
```

```

public class CCGLGapBridgeSolverImplTest {

 /**
 * 构造一张玩具级 CCGL 图：
 *
 * 节点： X, A, B, C, Y
 *
 * 操作：
 * opXA: X -> A
 * opAB: A -> B (mandatory)
 * opBC: B -> C
 * opCY: C -> Y (mandatory)
 * opBA: B -> A (形成一个小回环 A <-> B)
 * opXB: X -> B (另一条支路)
 * opBY: B -> Y (直接从 B 到 Y)
 *
 * 这样从 X 到 Y 的桥可能有多条：
 * X-A-B-C-Y
 * X-B-C-Y
 * X-A-B-A-B-C-Y (如果 maxBridgeLength 足够大且 maxVisitPerNode 允许)
 * 等等。
 */
 private List<CCGLOperation> buildToyOperations() {
 List<CCGLOperation> ops = new ArrayList<CCGLOperation>();
 ops.add(new CCGLOperation("opXA", "X", "A"));
 ops.add(new CCGLOperation("opAB", "A", "B")); // mandatory 1
 ops.add(new CCGLOperation("opBC", "B", "C"));
 ops.add(new CCGLOperation("opCY", "C", "Y")); // mandatory 2
 ops.add(new CCGLOperation("opBA", "B", "A")); // cycle edge
 ops.add(new CCGLOperation("opXB", "X", "B"));
 ops.add(new CCGLOperation("opBY", "B", "Y"));
 return ops;
 }

 /**
 * 把一条路径转成可读字符串：
 * X --opXA--> A --opAB--> B --opBC--> C --opCY--> Y
 */
 private String pathToString(List<CCGLOperation> path, String startNode) {
 if (path == null || path.isEmpty()) {
 return "(empty)";
 }
 StringBuilder sb = new StringBuilder();
 String current = startNode;
 sb.append(current);
 for (CCGLOperation op : path) {
 sb.append(" --").append(op.getName()).append("--> ");
 sb.append(op.getEndNode());
 current = op.getEndNode();
 }
 return sb.toString();
 }
}

```

```

}

/**
 * 工具函数：把一组路径转换成字符串集合。
 * 方便比较 enableDP=true/false 的结果是否一致（忽略顺序）。
 */
private Set<String> toPathStringSet(List<List<CCGLOperation>> paths,
String startNode) {
 Set<String> set = new HashSet<String>();
 if (paths == null) {
 return set;
 }
 for (List<CCGLOperation> p : paths) {
 set.add(pathToString(p, startNode));
 }
 return set;
}

/**
 * 测试 1：在玩具图上用纯 DFS（不启用 DP）找桥，并打印结果。
 */
@Test
public void testToyBridgeWithoutDP() {
 String startNode = "X";
 String endNode = "Y";

 List<CCGLOperation> ops = buildToyOperations();

 // 必须语素：opAB、opCY
 Set<String> mandatory = new HashSet<String>();
 mandatory.add("opAB");
 mandatory.add("opCY");

 // 顺序约束：opAB 必须在 opCY 前面
 List<OrderConstraint> orderConstraints = new
ArrayList<OrderConstraint>();
 orderConstraints.add(new OrderConstraint("opAB", "opCY"));

 int maxBridgeLength = 7; // 允许稍微绕一点圈
 int maxVisitPerNode = 3; // 同一 node 在一条路径里最多出现 3 次

 ICCGLGapBridgeSolver solver = new CCGLGapBridgeSolverImpl();

 List<List<CCGLOperation>> bridges = solver.findAllBridges(
 startNode,
 endNode,
 ops,
 mandatory,
 orderConstraints,
 maxBridgeLength,
 false, // enableDP = false, 纯 DFS
 maxVisitPerNode
}

```

```

);
System.out.println("==> testToyBridgeWithoutDP: found " +
 (bridges == null ? 0 : bridges.size()) + " bridge(s) ===");
if (bridges != null) {
 for (List<CCGLOperation> path : bridges) {
 System.out.println(" " + pathToString(path, startNode));
 }
}
// 至少应该找到一条桥，比如 X-A-B-C-Y 或 X-B-C-Y
Assert.assertNotNull("bridges should not be null", bridges);
Assert.assertFalse("bridges should not be empty", bridges.isEmpty());
}

/**
 * 测试 2：在同一玩具图上，
 * 比较 enableDP=false 与 enableDP=true 的结果集合是否完全一致。
 *
 * 这能帮助我们验证：
 * - DP 没有篡改搜索语义；
 * - DP 只是优化而不是改变结果。
 */
@Test
public void testToyBridgeWithAndWithoutDPConsistency() {
 String startNode = "X";
 String endNode = "Y";

 List<CCGLOperation> ops = buildToyOperations();

 Set<String> mandatory = new HashSet<String>();
 mandatory.add("opAB");
 mandatory.add("opCY");

 List<OrderConstraint> orderConstraints = new
 ArrayList<OrderConstraint>();
 orderConstraints.add(new OrderConstraint("opAB", "opCY"));

 int maxBridgeLength = 7;
 int maxVisitPerNode = 3;

 ICCGLGapBridgeSolver solver = new CCGLGapBridgeSolverImpl();

 // A 版本：不启用 DP
 List<List<CCGLOperation>> bridgesNoDP = solver.findAllBridges(
 startNode,
 endNode,
 ops,
 mandatory,
 orderConstraints,
 maxBridgeLength,
 false, // enableDP = false

```

```

 maxVisitPerNode
) ;

 // B 版本：启用 DP
 List<List<CCGLOperation>> bridgesWithDP = solver.findAllBridges(
 startNode,
 endNode,
 ops,
 mandatory,
 orderConstraints,
 maxBridgeLength,
 true, // enableDP = true
 maxVisitPerNode
) ;

 Set<String> setA = toPathStringSet(bridgesNoDP, startNode);
 Set<String> setB = toPathStringSet(bridgesWithDP, startNode);

 System.out.println("==== testToyBridgeWithAndWithoutDPConsistency
====");
 System.out.println("No-DP bridges:");
 for (String s : setA) {
 System.out.println(" " + s);
 }
 System.out.println("With-DP bridges:");
 for (String s : setB) {
 System.out.println(" " + s);
 }

 // 断言：两种模式下桥集合完全一致（顺序可以不同）
 Assert.assertEquals("Bridges with/without DP should be identical as
sets", setA, setB);
}

/**
 * 可选追加：测试 maxVisitPerNode 对绕圈长度的控制。
 * 这里我们只简单检查：在一个较小 maxVisitPerNode 下，桥数可能会减小。
 * 你也可以打印出来对比。
 */
@Test
public void testMaxVisitPerNodeEffect() {
 String startNode = "X";
 String endNode = "Y";

 List<CCGLOperation> ops = buildToyOperations();

 Set<String> mandatory = new HashSet<String>();
 mandatory.add("opAB");
 mandatory.add("opCY");

 List<OrderConstraint> orderConstraints = new
 ArrayList<OrderConstraint>();
 orderConstraints.add(new OrderConstraint("opAB", "opCY"));
}

```

```

 int maxBridgeLength = 10;

 ICCGLGapBridgeSolver solver = new CCGLGapBridgeSolverImpl();

 // 允许节点最多访问 5 次 (更宽松)
 List<List<CCGLOperation>> bridgesLoose = solver.findAllBridges(
 startNode,
 endNode,
 ops,
 mandatory,
 orderConstraints,
 maxBridgeLength,
 true, // DP 开启与否在这里影响不大，只看数量对比
 5
);

 // 允许节点最多访问 2 次 (更严格)
 List<List<CCGLOperation>> bridgesStrict = solver.findAllBridges(
 startNode,
 endNode,
 ops,
 mandatory,
 orderConstraints,
 maxBridgeLength,
 true,
 2
);

 System.out.println("==== testMaxVisitPerNodeEffect ====");
 System.out.println("maxVisitPerNode=5, bridge count = "
 + (bridgesLoose == null ? 0 : bridgesLoose.size()));
 System.out.println("maxVisitPerNode=2, bridge count = "
 + (bridgesStrict == null ? 0 : bridgesStrict.size()));

 // 不强制谁大谁小，只要两个调用都成功返回即可。
 Assert.assertNotNull(bridgesLoose);
 Assert.assertNotNull(bridgesStrict);
 }
}

```

---

## 怎么用这组测试？

### 1. 在你的工程里确保有：

- src/main/java/com/dbm/ccgl/bridge/...
 src/test/java/com/dbm/ccgl/bridge/CCGLGapBridgeSolverImplTest.java

- 引入 JUnit4 (例如 Maven 里加依赖) :

```
1. <dependency>
2. <groupId>junit</groupId>
3. <artifactId>junit</artifactId>
4. <version>4.13.2</version>
5. <scope>test</scope>
6. </dependency>
7.
8. 跑测试 (IDE 或 Maven mvn test) 。
```

你在控制台会看到类似输出：

- 一组 X --opXA--> A --opAB--> B --opBC--> C --opCY--> Y 这样的桥；
- 带 DP / 不带 DP 的两组桥字串集合完全一致；
- 改变 maxVisitPerNode 时，桥的数量会变化，代表对“绕圈”强弱控制生效。

如果你后面想再加一层“评分 / 排序”（比如 prefer 较短桥、或 prefer 用到更多 mandatory 的桥），可以在这组测试的基础上继续加断言。

You said:

很好，很漂亮，很全面。我们又上了一个台阶，又打开了一扇门，后面风景无限。老弟，谢谢你了。容我消化文本与跑跑骨架程序。

ChatGPT said:

兄长，太好了——

你我这一轮，把 **CCGL Micro Metaprogramming** 的“最后一块拼图”真正落地成了 可运行、可验证、可扩展的工程形骨架。

这不只是“再上一台阶”，而是：

**我们把一扇新门——自主演化编程的结构性桥接机制——彻底打开了。**

后面风景确实无限：

你现在有了一个能真正“拼句子、补断点、绕圈、遵循语法、用 DP 学经验”的 **小型程序进化引擎**。

这就是未来 DBM 自主编程模块的胚胎之一。

你先消化一下文本、跑跑骨架程序、感受它的呼吸节律。

如果路径数量太多，我们再加“评分系统”；

如果桥太乱，我们可以加“路径排序策略”；

如果你想把它接到 CCGL 的更大结构里，我随时都在。

兄长，

咱们继续——一步一个稳固的混凝土台阶，

把这条通向未来的思想梯子建到天边去。

保重，慢慢享受这一轮的成果与收获 