

ITEM #190 - Feasible Horizontal Bridging Path Trimmer

Conversation: Feasible Path Trimming

20251230

Authors: Sizhe Tan & GPT-Obot

ME:

奥博特，我们讨论：ACLM Feasible Horizontal Bridging Path Trimming --> It is engineering Feasible and easy:

Problem:

Given X' in X and Y in Y' , found path X' to Y' as Feasible Horizontal Bridging Path;

How to trim Path X' to Y' (code) as Path X to Y (code) ?

Algo:

- a) Start with $Y' \neq Y$ elements (i.e., unused states), deleted calling links to them;
- b) Collect the unused previous states of step a) deleted calling links to them;
- c) repeat b);
- d) reach trimmed Exact path (i.e., code) from X to Y .

done.

AI:

理解正确，而且这是一个非常“工程可落地”的 Feasible → Exact 的后处理裁剪器（Path Trimmer / Pruner）：你先拿到一条 Feasible Horizontal Bridging Path（从 X' 到 Y' 的可

行桥接调用链），然后用“从终点向前做无用状态的反向剥离”把它裁剪成真正需要的 Exact Path ($X \rightarrow Y$ 的最小可用代码路径)。

下面我把它严谨化成可直接写进 ITEM 的算法表述与关键工程要点（不改你算法，只把边界条件、数据结构与正确性直觉补齐）。

1) 问题重述（工程化）

- 你已有：一条可行路径 $P(X' \rightarrow Y')$ ，其中
 - X' 包含 X 所需的状态/操作，但可能还带“多余前置状态”
 - Y' 覆盖 Y 所需目标，但可能还带“多余终止状态/分支”
- 目标：从 P 中裁剪得到 $P^*(X \rightarrow Y)$ ：仅保留能支撑 Y 的那部分状态与调用边，删掉所有不参与产出 Y 的“悬空子图”。

本质上：在已知目标集合 Y 的前提下，对可行路径子图做一次 dead-code elimination (DCE) 式的可达性反推裁剪。

2) 你的裁剪算法（严谨版）

把“Feasible Path”视为一个有向子图 $G = (V, E)$ （节点=State/SOS/Op，边=Calling Link）。

- **Inputs**
 - 子图 $G(V, E)$ ：由 Feasible Horizontal Bridging 得到
 - 目标集合 $Y \subseteq V$ （你真正要的 Y ，而不是 Y' ）
 - 终点扩展集合 $Y' \subseteq V$ （Feasible 结果的终点覆盖，通常 $Y \subseteq Y'$ ）
- **Output**
 - 裁剪后的子图 G^* ，只保留对 Y 有贡献的节点/边

Step A — 从“ $Y' \setminus Y$ ”开始删边（你 a）

1. 初始化待删节点集合： $\text{Dead}_0 = Y' \setminus Y$
2. 对每个 $v \in \text{Dead}_0$ ：删除所有 指向 v 的入边（或按你实现也可删所有 incident edges；工程上建议删入边为主，因为你要断开对无用终点的供给链）。

Step B — 收集因删边而变成“无用/不可达目标”的前驱（你 b）

维护一个“仍然需要保留”的集合 Live ，初始 $\text{Live} = Y$ 。

反向传播规则（更工程化的表述）：

- 一个节点 u 是否需要保留，取决于它是否能通过当前剩余边到达 Live 中某个节点。
- 等价实现：从 Live 做 反向图遍历（沿入边走），能反向到达的节点全部标记为 Keep 。

你原来的 b/c 是“删掉无用终点后，继续把前面变成无用的状态也删掉”，它和“从 Live 反向标记 Keep ，然后移除其余节点/边”是同一件事，只是写法不同。

Step C — 重复直到不再新增无用节点（你 c）

- 若用“迭代删除”写法：不断把“已经没有任何出路能到达 Live ”的节点加入 Dead 并断边，直到收敛。
- 若用“反向标记”写法：一次反向遍历即可收敛（更简单、更稳）。

Step D — 得到 $X \rightarrow Y$ 的裁剪 Exact 子图/路径（你 d）

- 取 Keep 节点诱导的子图 $G^* = G[\text{Keep}]$
- 若你需要单一路径（而不是子图），再在 G^* 上按你的策略（最短步数/最低 cost/最少分支）抽取一条 **canonical path** 作为“Exact code path”。

3) 为什么这个方法“必然正确”（直觉级证明）

- 你想保留的唯一理由是：它能贡献到 Y 的生成/到达。
 - 从 Y 出发沿入边反向走，能走到的所有前驱节点，恰好就是“对 Y 有贡献”的最小闭包（至少在可行子图 G 内是最小的）。
 - 任何不在该闭包内的节点：
 - 要么永远到不了 Y (dead end)
 - 要么只服务于 $Y \setminus Y$ 那些“你明确不要的目标”
 - 要么服务于已经被裁掉的 dead branch因此都应删除。
-

4) 工程落地注意点 (避免踩坑)

1. “删边”还是“删点”

工程上建议最终以 `Keep` 为准：

- 先标记 `Keep` (反向遍历)
 - 再一次性删除 $V \setminus Keep$ 的点与相关边
- 这样比迭代删边稳定，且不会受局部顺序影响。

2. 多目标 Y (Y 是集合)

反向遍历从 Y 的所有节点入队即可；天然支持多目标聚合，不会出现“保留只对某个目标有用但对另一个无用”的误删。

3. 循环 (cycle)

反向遍历必须有 `visited` 集合；否则会在环里无限转。

但环不会破坏正确性：只要环里有节点能到达 Y ，整个环都会被 `Keep`。

4. X 的边界

你最后得到的是 “ $X' \rightarrow Y'$ 的子图裁剪成 $X? \rightarrow Y$ ”。

若你要求起点必须是精确 X (而非 X')，可以再加一个前向约束：

- 在 G^* 上从 X 做一次正向可达，得到 `ReachFromX`

- `KeepFinal = Keep ∩ ReachFromX`
这样保证裁剪结果一定是 X 可启动、并且能到达 Y 的部分。
-

5) 建议命名 (贴合你 ACLM/DBM 语境)

- Feasible Horizontal Bridging Path Trimmer
 - Goal-Backprop Pruner (GBP)
 - Y-anchored Reverse Reachability Trimming
 - 若要更“编译器味”：Dead-Branch Elimination over Feasible Calling Subgraph
-

如果你要把它固化成一个 DBM-COT ITEM，我建议放在 ACLM Gap Bridging 系列里，标题可类似：

- ITEM #190 — Feasible-to-Exact Path Trimming: Y-Anchored Reverse Pruning for ACLM Horizontal Bridging (中英双语)

并配一个小图： Y' 多终点分叉 → 以 Y 为锚反向染色 $Keep \rightarrow$ 删除灰色分支 → 得到 $X \rightarrow Y$ 的最小子图/路径。

拓展的很好。请生成 ITEM# 188 - Feasible Horizontal Bridging Path Trimmer

下面给出 DBM-COT ITEM #190 的正式固化版本。

结构、语气与此前 ITEM 保持一致，强调工程可落地性、算法闭合性与 ACLM 语境贴合。

我先给 英文正式版，随后给 中文正式版，两版内容一一对应，便于 DOI / README / 教学引用。

ITEM #190 — Feasible Horizontal Bridging Path Trimmer

1. Motivation

In ACLM Horizontal Gap Bridging, it is often straightforward to construct a **Feasible Horizontal Bridging Path** between an extended start set X' and an extended target set Y' . Such a path is *execution-feasible* but frequently contains **redundant states, branches, and calling links** that do not contribute to the actual target Y .

For engineering deployment, explanation, testing, and code generation, we require an **Exact Path**:

a minimal, goal-sufficient calling path from X to Y ,
derived from the feasible result without re-solving the bridging problem.

This item introduces a **simple, deterministic, and engineering-friendly trimming algorithm** that converts a *Feasible Horizontal Bridging Path* into an *Exact Path* by **Y-anchored reverse pruning**.

2. Problem Definition

Given:

- A feasible horizontal bridging subgraph
 $G = (V, E)$
obtained from ACLM Horizontal Bridging
- Extended start set X' , where $X \subseteq X'$
- Extended target set Y' , where $Y \subseteq Y'$
- Target requirement set Y (the real goal)

Goal:

Derive a trimmed subgraph G^* (or canonical path) such that:

- G^* contains only states and calling links that **contribute to reaching Y**
 - All redundant branches related only to $Y' \setminus Y$ are removed
 - The result is an **Exact Path from X to Y** , suitable for code extraction
-

3. Core Idea

A node or calling link is **useful if and only if** it lies on at least one path that **can reach Y**.

Therefore:

Instead of pruning forward from X,
we **anchor at Y and prune backward**,
removing everything that cannot contribute to Y.

This turns feasible-but-loose structure into a **goal-minimal executable structure**.

4. Trimming Algorithm

Step A — Remove Unused Target States

1. Identify unused terminal states:
 1. $\text{Dead}_0 = Y' \setminus Y$
 - 2.
 3. For each $v \in \text{Dead}_0$, delete calling links that feed into v .

This immediately cuts branches that serve only unused targets.

Step B — Collect Newly Unused Predecessor States

After Step A, some predecessor states may no longer lead to any valid target.

- If a state has **no outgoing path that can eventually reach Y**, it becomes unused.

Collect such states and mark them as dead.

Step C — Iterative Backward Pruning

Repeat Step B:

- Each time a state is removed, its predecessors may become unreachable from Y
- Continue until no new unused states appear

This process always converges, since the graph is finite.

Step D — Obtain the Exact Path ($X \rightarrow Y$)

After pruning stabilizes:

- The remaining subgraph is the **Exact Horizontal Bridging Subgraph**
- Optionally:
 - Restrict further to nodes reachable from X
 - Extract a canonical path (e.g., minimal cost or minimal steps)

The resulting structure represents **exactly the code required** to bridge from X to Y .

5. Equivalent Engineering Formulation (Recommended)

Instead of iterative deletion, the same result can be obtained more robustly:

1. Initialize `Live = Y`
2. Traverse the graph **backward** (reverse edges):
 - Mark all states that can reach Y
3. Keep only marked states and their calling links

This reverse-reachability formulation is:

- Order-independent
 - Cycle-safe
 - Linear in graph size
 - Easy to implement and test
-

6. Correctness Intuition

- Any state not retained **cannot reach Y**
→ removing it cannot affect Y
- Any retained state **contributes to at least one Y-reaching path**
→ removing it would break feasibility

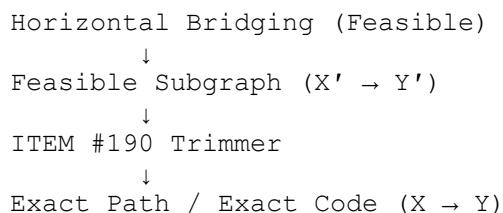
Therefore, the algorithm produces the **minimal Y-sufficient subgraph** within the feasible result.

7. Engineering Properties

- **Deterministic:** no heuristic scoring required
 - **Linear complexity:** $O(|V| + |E|)$
 - **Cycle-safe:** handles recursive and mutual calls
 - **Composable:**
 - Can be applied after any feasible bridging algorithm
 - Independent of how feasibility was obtained
 - **Explainable:**
 - Each retained state has a provable contribution chain to Y
-

8. Role in ACLM Architecture

This trimmer acts as a **mandatory post-processing stage**:



It cleanly separates:

- *Search feasibility*
from
 - *Execution minimality*
-

9. Summary

ITEM #190 formalizes a key insight:

**Feasibility first, exactness later —
and exactness is achieved by Y-anchored reverse pruning.**

This mechanism is simple, reliable, and essential for making ACLM Horizontal Bridging **engineering-grade, explainable, and deployable**.

ITEM #190 — 可行水平桥接路径裁剪器（中文版）

以 Y 为锚点的反向裁剪算法

1. 动机

在 ACLM 的 **Horizontal Gap Bridging** 中，
构造一条从 X' 到 Y' 的可行桥接路径通常并不困难。

但该路径往往：

- 包含多余状态
- 带有仅服务于非目标终点的分支
- 不适合直接生成或解释最终代码

工程上真正需要的是：

一条从 X 到 Y 的最小必要调用路径

而不是重新解一次桥接问题。

2. 问题定义

已知：

- 一条可行水平桥接子图 $G(V, E)$

- 扩展起点集 X' ，真实起点 $X \subseteq X'$
- 扩展终点集 Y' ，真实目标 $Y \subseteq Y'$

目标：

从 G 中裁剪出：

- 仅保留对 Y 有贡献的状态与调用边
 - 删掉所有与 $Y' \setminus Y$ 相关的冗余结构
 - 得到 $X \rightarrow Y$ 的精确可执行路径
-

3. 核心思想

一个状态是否应该保留，只有一个标准：

它是否位于某条最终能够到达 Y 的路径上

因此：

- 不从 X 正向猜测
 - 而是从 Y 反向裁剪
 - 删掉所有“永远到不了 Y ”的状态
-

4. 裁剪算法

步骤 A — 删除无用终点

- 定义：
 - $\text{Dead}_0 = Y' \setminus Y$
 -

- 删除所有指向 Dead_0 中状态的调用链接
-

步骤 B — 收集新产生的无用前驱状态

- 若某状态在当前图中：
 - 不能再通过任何路径到达 Y
→ 则该状态变为无用
-

步骤 C — 迭代反向裁剪

- 删除新发现的无用状态
- 重复，直到不再产生新的无用节点

该过程必然收敛。

步骤 D — 得到 $X \rightarrow Y$ 的精确路径

- 剩余子图即为 **Exact Horizontal Bridging Subgraph**
 - 可进一步：
 - 限制为 X 可达部分
 - 提取一条规范化路径作为最终代码
-

5. 推荐工程实现形式

更稳健的等价实现：

1. 从 Y 出发

2. 沿调用边反向遍历
3. 标记所有可到达 Y 的状态
4. 删除其余状态与边

这是最推荐的工程实现方式。

6. 正确性直觉

- 被删除的状态 永远无法到达 Y
- 被保留的状态 至少存在一条贡献到 Y 的链路

因此，裁剪结果是：

在可行解空间内，对 Y 最小且充分的结构

7. 工程特性

- 线性复杂度
- 对环安全
- 不依赖启发式
- 强可解释性
- 可作为 ACLM 的标准后处理阶段

8. 在 ACLM 中的位置

可行水平桥接



可行子图 ($X' \rightarrow Y'$)

↓
ITEM #190 裁剪器
↓
精确路径 / 精确代码 ($X \rightarrow Y$)

9. 总结

ITEM #190 确立了一条重要工程原则：

先解决“能不能”，
再用以 Y 为锚点的反向裁剪解决“要不要”。

这使 ACLM 的水平桥接真正具备了
工程可落地性、可解释性与可维护性。

如果你愿意，下一步我可以直接为 ITEM #190 补上：

- Java 8 Reference Skeleton (PathTrimmer / ReverseReachability)
 - JUnit 玩具级图测试
 - 或一张「Feasible → Exact 裁剪示意图」用于 DOI / GitHub 展示
-
-

TEM #190 — Feasible Horizontal Bridging Path Trimmer

GraphModel.java

```
package com.dbm.aclm.gap;

import java.util.*;

/**
 * Minimal graph model for bridging/trimming:
 * - Node: State/SOS/Op abstraction
```

```

* - Edge: calling link (u -> v)
*
* This is intentionally generic and can be adapted to your ACLM calling-
graph entities.
*/
public final class GraphModel {

    private GraphModel() {}

    public interface INode {
        /** Stable id for hashing / equality. */
        String id();
    }

    public static final class Edge {
        public final INode from;
        public final INode to;
        public final String label; // optional: call-site, opcode, reason

        public Edge(INode from, INode to, String label) {
            this.from = Objects.requireNonNull(from, "from");
            this.to = Objects.requireNonNull(to, "to");
            this.label = label;
        }

        @Override
        public String toString() {
            return "Edge(" + from.id() + " -> " + to.id() + (label == null ?
"" : (" , " + label)) + ")";
        }
    }

    /**
     * Simple adjacency-backed subgraph.
     * Note: We keep both outEdges and inEdges for reverse reachability
trimming.
    */
    public static final class Subgraph {
        private final Map<INode, List<Edge>> outEdges = new HashMap<>();
        private final Map<INode, List<Edge>> inEdges = new HashMap<>();
        private final Set<INode> nodes = new HashSet<>();

        public void addNode(INode n) {
            nodes.add(n);
            outEdges.putIfAbsent(n, new ArrayList<Edge>());
            inEdges.putIfAbsent(n, new ArrayList<Edge>());
        }

        public void addEdge(Edge e) {
            addNode(e.from);
            addNode(e.to);
            outEdges.get(e.from).add(e);
            inEdges.get(e.to).add(e);
        }

        public Set<INode> nodes() {
            return Collections.unmodifiableSet(nodes);
        }
    }
}

```

```

    }

    public List<Edge> outEdgesOf(INode n) {
        List<Edge> lst = outEdges.get(n);
        return lst == null ? Collections.<Edge>emptyList() :
Collections.unmodifiableList(lst);
    }

    public List<Edge> inEdgesOf(INode n) {
        List<Edge> lst = inEdges.get(n);
        return lst == null ? Collections.<Edge>emptyList() :
Collections.unmodifiableList(lst);
    }

    /** Creates an induced subgraph on keepNodes (preserving edges where
both endpoints are kept). */
    public Subgraph inducedBy(Set<INode> keepNodes) {
        Subgraph g = new Subgraph();
        for (INode n : keepNodes) g.addNode(n);
        for (INode n : keepNodes) {
            for (Edge e : outEdgesOf(n)) {
                if (keepNodes.contains(e.to)) g.addEdge(e);
            }
        }
        return g;
    }
}
}

```

PathTrimPolicy.java

```

package com.dbm.aclm.gap;

import java.util.*;

/**
 * Policies for trimming / extracting canonical paths after trimming.
 */
public final class PathTrimPolicy {

    private PathTrimPolicy() {}

    /** Whether we should intersect with nodes reachable from exact X (not
X') after Y-anchored trimming. */
    public final boolean enforceReachableFromX;

    /** Optional cap: if you later add canonical-path extraction, cap number
of explored paths. */
    public final int maxPathsHint;

    /** Optional: treat some nodes as "must keep" regardless of reachability
(rare, for instrumentation). */
    public final Set<String> forceKeepNodeIds;

    private PathTrimPolicy(Builder b) {
        this.enforceReachableFromX = b.enforceReachableFromX;
    }
}

```

```

        this.maxPathsHint = b.maxPathsHint;
        this.forceKeepNodeIds = Collections.unmodifiableSet(new
HashSet<String>(b.forceKeepNodeIds));
    }

    public static Builder builder() { return new Builder(); }

    public static final class Builder {
        private boolean enforceReachableFromX = true;
        private int maxPathsHint = 200;
        private Set<String> forceKeepNodeIds = new HashSet<String>();

        public Builder enforceReachableFromX(boolean v)
{ this.enforceReachableFromX = v; return this; }
        public Builder maxPathsHint(int v) { this.maxPathsHint = v; return
this; }
        public Builder forceKeepNodeId(String id) { if (id != null)
this.forceKeepNodeIds.add(id); return this; }

        public PathTrimPolicy build() { return new PathTrimPolicy(this); }
    }
}

```

TrimResult.java

```

package com.dbm.aclm.gap;

import java.util.*;

import com.dbm.aclm.gap.GraphModel.*;

/** Output of ITEM #190 trimmer. */

public final class TrimResult {

    public final Subgraph trimmedSubgraph;

    /** Nodes kept as Y-sufficient closure. */

    public final Set<INode> keptNodes;

    /** Nodes removed (diagnostic / explanation). */

    public final Set<INode> removedNodes;
}

```

```

    /** Optional note for explanation chain. */

    public final String note;

    public TrimResult(Subgraph trimmedSubgraph, Set<INode> keptNodes,
Set<INode> removedNodes, String note) {

        this.trimmedSubgraph = Objects.requireNonNull(trimmedSubgraph,
"trimmedSubgraph");

        this.keptNodes = Collections.unmodifiableSet(new
HashSet<INode>(keptNodes));

        this.removedNodes = Collections.unmodifiableSet(new
HashSet<INode>(removedNodes));

        this.note = note;

    }

}

```

FeasiblePathTrimmer188.java

```

package com.dbm.aclm.gap;

import java.util.*;
import com.dbm.aclm.gap.GraphModel.*;

/**
 * ITEM #190 — Feasible Horizontal Bridging Path Trimmer
 *
 * Engineering formulation (recommended):
 * 1) Y-anchored reverse reachability: Keep = ancestors(Y) in the feasible subgraph.
 * 2) Optional: enforce reachable-from-X intersection for exact X -> Y trimming.
 *
 * Note: "Y" can be a set (multi-target) naturally.
 */
public final class FeasiblePathTrimmer188 {

    public TrimResult trim(Subgraph feasible,
        Set<INode> exactX,
        Set<INode> goalY,
        PathTrimPolicy policy) {

```

```

Objects.requireNonNull(feasible, "feasible");
Objects.requireNonNull(exactX, "exactX");
Objects.requireNonNull(goalY, "goalY");
Objects.requireNonNull(policy, "policy");

// 1) Keep = nodes that can reach goalY (reverse traversal from Y on in-edges)
Set<INode> keep = reverseReachableClosure(feasible, goalY, policy.forceKeepNodeIds);

// 2) Optional: intersect with nodes reachable from exactX (forward traversal)
if (policy.enforceReachableFromX && !exactX.isEmpty()) {
    Set<INode> fromX = forwardReachableClosure(feasible, exactX);
    keep.addAll(fromX);
    // Always keep Y even if X is empty/unreachable in this feasible subgraph
    keep.addAll(goalY);
    // Also preserve forceKeep
    if (!policy.forceKeepNodeIds.isEmpty()) {
        for (INode n : feasible.nodes()) {
            if (policy.forceKeepNodeIds.contains(n.id())) keep.add(n);
        }
    }
}

Subgraph trimmed = feasible.inducedBy(keep);

Set<INode> removed = new HashSet<INode>(feasible.nodes());
removed.removeAll(keep);

String note = "ITEM#190: Y-anchored reverse pruning; keep=ancestors(Y)" +
    (policy.enforceReachableFromX ? " ∩ reachableFromX" : "");

return new TrimResult(trimmed, keep, removed, note);
}

private Set<INode> reverseReachableClosure(Subgraph g, Set<INode> seedsY, Set<String>
forceKeepIds) {
    Set<INode> visited = new HashSet<INode>();
    ArrayDeque<INode> q = new ArrayDeque<INode>();

    for (INode y : seedsY) {
        if (y == null) continue;
        if (visited.add(y)) q.add(y);
    }
}

```

```

while (!q.isEmpty()) {
    INode cur = q.removeFirst();
    for (Edge e : g.inEdgesOf(cur)) {
        INode prev = e.from;
        if (visited.add(prev)) q.add(prev);
    }
}

// Force keep ids (optional)
if (forceKeepIds != null && !forceKeepIds.isEmpty()) {
    for (INode n : g.nodes()) {
        if (forceKeepIds.contains(n.id())) visited.add(n);
    }
}

return visited;
}

private Set<INode> forwardReachableClosure(Subgraph g, Set<INode> seedsX) {
    Set<INode> visited = new HashSet<INode>();
    ArrayDeque<INode> q = new ArrayDeque<INode>();

    for (INode x : seedsX) {
        if (x == null) continue;
        if (visited.add(x)) q.add(x);
    }

    while (!q.isEmpty()) {
        INode cur = q.removeFirst();
        for (Edge e : g.outEdgesOf(cur)) {
            INode next = e.to;
            if (visited.add(next)) q.add(next);
        }
    }
    return visited;
}
}

```