

ITEM #234 - Hybrid Early-Stop Acceleration for UnalignedAND-BTP (Large-vs-Large)

4A + 4B Head Capture → 2B Composition Top-K → Optional Symmetry Pass

Conversation : UnalignedAND-BTP 算法加速

20260113

Authors: Sizhe Tan & GPT-Obot

ME :

奥博特，我们讨论：再论 UnalignedAND-BTP 算法加速

1。昨天我们进行了一轮 UnalignedAND-BTP 算法加速的系统讨论。我有一点意犹未尽。主要疑问是是：

** 这是一个 Optimization Problem With Long Tail （长尾优化问题）；

** 我们昨天的讨论似乎倾向有不记工本追求与寻找长尾长尾中的低收益局部解；

** 这比较违背 DBM 的最小进化门槛算法理念。

2。因此，我们在取得足够置信度的前提下，应该考虑比较能及时收手的算法。比如，

3。UnalignedAND-BTP 算法加速的 Hybrid 集合算法 to Improve Large Starmap VS Large Starmap UnalignedAND-BTP:

Step 1:

Use Part of Algo 5A) Human Genomic Project like Approach:

- a) Cut Large StarmapA into small pieces;
- b) For each peach of LargeStarmapA,
 - b1) call above algo 4A) to find small-to-large matches with Large StarmapB;
 - b2) collect all matches.
- c) Repeat as necessary to collect more matches.

Step 2:

Use Part of 5B) Anchor Search Segmentation Tier-wise Matching/ZoomingIn Approach:

- a) Cut Large StarmapA into small pieces;
- b) For each peach of LargeStarmapA,
 - b1) call above algo 4B) to find small-to-large matches with Large StarmapB;
 - b2) collect all matches.
- c) Repeat as necessary to collect more matches.

Step 3:

- a) Merge found matches of Step 1 and Step 2
i.e., the merged Known Matched SubPattern of LargeStarmapB that match LargeStarmapA partially.
- b) Use Algo 2B)
 - Given a number of known Metric Space Points, predict a large Metric Space Point (i.e., stracture)
as combination of the known Metric Space Points. e.g. · 生物蛋白结构预测 ·

to find top-K matches of LargeStarmapA as non-conflicting combinations of the merged Known Matched SubPattern.

These top-k matches will be the top-K matches of LargeStarmapA and LargeStarmapB.

4. If we want Further Zooming-in the coverage and confidence score, just,

a) switch LargeStarmapA as LargeStarmapB, and LargeStarmapB as LargeStarmapA;

b) rerun algo 3, to find top-k matches of two;

c) merge two top-K matches of 3) and 4) and eliminated duplicated ones.

done.

请综述·评论与拓展。

Previous Prompt :

奥博特，我们讨论：UnalignedAND-BTP(BucketTreeOfPermutation)的加速算法

1。UnalignedAND-BTP 是 DBM Metric Distance 的核心算法。对 GraphStarmap, SequenceStarmap, ImageStarmap

等非简单 Metric Space 结构来说，求可控精度解 (non-heuristic-solution) 而不可绕过的重要工具。

2。UnalignedAND-BTP 可解两类问题：

2A) Find Two Metric Space Points' Best Match or Top K Matches。e.g.·Metric Distance;

2B) Given a number of known Metric Space Points, predict a large Metric Space Point (i.e., stracture) as combination of the known Metric Space Points.
e.g.·生物蛋白结构预测。

3。UnalignedAND-BTP 的 Computation Complexity 是较大的。尤其当一

个或两个 Metric Space Point 所含的 Nodes 数量很大的时候。

任何能改善与加速 UnalignedAND-BTP 算法的工程与算法方式都是极其欢迎与意义重大的。我们把改进分两类：

3A) Improve Small StarmapA VS Large StarmapB
UnalignedAND-BTP;

3B) Improve Large StarmapA VS Large StarmapB
UnalignedAND-BTP;

4. For Problem 3A) - Improve Small Starmap VS Large Starmap
UnalignedAND-BTP:

Two Approaches:

4A) Human Genomic Project like Approach:

a) Cut Large StarmapB into pieces, each pieces
is larger than 3 times of Small StarmapA
to have enough room to avoid missing matches
in border area of pieces;

b) Find matches for each pair of SmallStarmapA
VS LargeStarmapBPieces;

c) Collect all matches and eliminate duplicated
matches;

done.

4B) Anchor Search Segmentation Tier-wise
Matching/ZoomingIn Approach:

a1) In Small StarmapA, select some Anchors
Nodes, and search Large StarmapB for anchor nodes's
occurrences.

a2) At each of anchor nodes' occurrences, cut a
large enough piece. Say LargeStarmapBPieces;

b) Find matches for each pair of SmallStarmapA
VS LargeStarmapBPieces;

c) Collect all matches and eliminate duplicated

matches;

d) Repeat a) to c) until Anchors Nodes are not attractive enough anymore;

done.

5. For Problem 3B) - Improve Large Starmap VS Large Starmap UnalignedAND-BTP:

5A) Human Genomic Project like Approach:

a) Cut Large StarmapA into small pieces;

b) For each peach of LargeStarmapA,

b1) call above algo 4A) to find small-to-large matches with Large StarmapB;

b2) collect all matches

d) For each match in the collected all matches in b),

if the match has no overlap with any other match and

if the match has no node at LargeStarmapA piece's border, (**** very important to enable to catch border match(s) in subsequent process)
simplify both LargeStarmapA and LargeStarmapB by removing all nodes umples by this match;

e) Repeat a) to d) until no more match can be collected.

f) Finally, use algo 3A) to find matches of the final reduced LargeStarmapA and LargeStarmapB and collect the matches.

done.

5B) Anchor Search Segmentation Tier-wise Matching/ZoomingIn Approach:

a) Cut Large StarmapA into small pieces;

b) For each peach of LargeStarmapA,

```

        b1) call above algo 4B) to find small-
to-large matches with Large StarmapB;
        b2) collect all matches

    d) For each match in the collected all matches
in b),
        if the match has no overlap with any
other match and
            if the match has no node at
LargeStarmapA piece's border,  (**** very important to
enable to catch border match(s) in subsequent process)
                simplify both LargeStarmapA and
LargeStarmapB by removing all nodes umples by this match;

    e) Repeat a) to d) until no more match can be
collected.

    f) Finally, use algo 3A) to find matches of the
final reduced LargeStarmapA and LargeStarmapB
        and collect the matches.

done.

```

奥博特，这套算法非常复杂，逻辑嵌套回转的很，有问题我当进一步誊清。

请评论与拓展。

AI：

你的担心是成立的：**Large-vs-Large 的 UnalignedAND-BTP 加速确实是长尾优化问题**。如果按“逐层 zoom-in、逐步保证召回、逐步榨干边角收益”的路线走，很容易落入“长期消耗工本追低收益局部改进”的陷阱，这与 DBM 的 **最小进化门槛（MET）** 理念不一致。

你提出的 Hybrid（4A + 4B → 2B 组合求解 → 可选对称再跑一轮）是一个非常 DBM 风味的“及时收手”方案：**先把可获得的高置信局部块抓出来，然后把剩下的全局匹配问题降维**

成“组合一致性/无冲突选择”，在可控时间内给出 top-K 结构解释。下面我按“理论逻辑、正确性边界、工程落地、可扩展增强”来综述与拓展。

1) 你这个 Hybrid 的本质：把长尾从“匹配空间”搬到“组合空间”

传统 RTB（切片/锚定/CCC/peel/迭代）是：

- 在 BTP 的匹配空间里不断细化召回与验证，长尾集中在“更多候选片段 + 更深搜索”。

你这个 Hybrid 是：

- 用 4A 与 4B 尽快抓取一批高质量的局部已匹配子模式（Known Matched SubPatterns）；
- 然后用 2B 的“结构拼装”求解器，把剩余问题转成：
从这些子模式里挑出一组不冲突组合，使其覆盖/解释 LargeStarmapA 的 top-K。

这是一种典型的 DBM “先锁住结构骨架，再做细节”的策略，和 MET 非常一致：

- 早期收益大（抓到最明显的子结构）；
 - 后续收益迅速递减时就可以停（组合解已足够解释大局）。
-

2) 对 Step 1/2 的评价：4A 与 4B 是互补的“高收益头部捕获器”

Step 1（4A 全覆盖切片）

优点：**Exact-friendly**，coverage 可控，不容易漏掉“明显且分布均匀”的匹配。

缺点：成本较稳但可能偏高；在某些结构上会抓到一堆“弱相关”块。

Step 2 (4B anchor+CCC)

优点：能快速命中“强区分度局部结构”，对重复模式少的区域非常高效。

缺点：若过度过滤会漏解，因此 **EXACT 模式下应更多使用 DEFER，而不是 REJECT**；

RECALL 模式要输出 coverage/miss-risk。

组合使用的意义：

- 4A 提供“兜底覆盖”，避免纯 anchor 召回导致的盲区；
- 4B 提供“稀疏高置信块”，显著提升头部收益并减少后续组合复杂度。

你在 Step 1/2 都写了“Repeat as necessary”——这里建议用 MET 风格加一个硬停止准则（见第 5 节）。

3) Step 3 是关键：2B 组合求解把“大对大匹配”降维成“无冲突拼装”

你说的“这些 top-k matches will be the top-K matches of LargeStarmapA and LargeStarmapB”在逻辑上成立，但需要一个明确的前提，否则会变成“看起来像”的结论：

3.1 成立的前提（必须写清楚）

要让“组合 top-K == 全局 top-K”，至少需要满足其一：

- **P1：完备候选前提（强）**

你的 Step1+Step2 抓到的子模式集合 包含了全局最优解所使用的所有必要子结构块（或至少包含一个能导向最优的等价块集合）。

- **P2：可证明下界前提 (MET-friendly)**

对“未被捕获的部分”有一个可计算的最小剩余成本下界 **LB_remain**，使得：

`best_combo_cost + LB_remain >= current_topK_threshold`

从而可以证明继续深入不可能改变 top-K。

如果没有这些前提，Hybrid 更准确的定位是：

- 一个高置信的 top-K 近似/候选集生成器，并且可以通过对称再跑、回补 deferred、提高覆盖率来逐步逼近 exact。

这不削弱它的价值；反而更贴合“及时收手 + 可递进增强”的工程哲学。

3.2 2B 组合求解器应当具备的 Contract

为了让 Step3 “可控时间收敛”，2B 组合求解器需要至少三项能力：

1. **Conflict Graph**：子模式之间冲突（共享 B 节点、共享 A 区域、映射不一致）。
2. **Coverage / Benefit**：每个子模式覆盖 A 的哪些需求（tiers/segments/motifs）。
3. **Objective**：组合的总成本（距离） + 未覆盖惩罚（或下界）。

这实际上是一个“加权集合打包/最大兼容集合”问题，但你不需要最优解：

- 你要的是 top-K 且 可早停，所以 branch-and-bound + LB_remain 极其合适。

4) Step 4（对称互换再跑一轮）是很强的“低成本增信”手段

你提出的对称迭代，本质上是在做：

- 从 $A \rightarrow B$ 得到一组组合解释；
- 再从 $B \rightarrow A$ 得到另一组组合解释；
- 合并去重，得到更稳健的交集/并集。

这在复杂结构匹配里非常实用，因为 A 与 B 往往存在“结构密度不均、锚点稀有度不均”的不对称性。

对称跑一轮的价值是：

- 提升覆盖盲区的概率；
- 用“互相证实”强化置信度评分；
- 给出一个非常 DBM 的解释输出：哪些块是双向一致的（高置信），哪些只是一侧提出（待复核）。

建议你把 Step 4 明确成一个可选开关：`symmetryPass = {OFF, ONCE, UNTIL_STABLE}`，默认 ONCE，符合 MET。

5) 你要的“及时收手”：给 Hybrid 加三条 MET 级停止准则

这是你这次问题的核心诉求。我建议直接固化三条“可实现、可解释”的 stop rules：

Stop Rule S1：边际收益阈值

每一轮 Step1/2 增加的新子模式集合，若带来的：

- 覆盖增量 (`coveredMass`) $< \epsilon_{cov}$
或
- top-K 组合成本改善 $< \epsilon_{cost}$
则停止继续收集更多 matches。

Stop Rule S2：组合下界封顶

维护一个 `LB_remain`（未覆盖部分的最小可能成本/惩罚下界）。当：

- `best_combo_cost + LB_remain >= kth_combo_cost`
并且

- deferred/backfill 的潜在改善上界不足
则停止（Top-K 稳定）。

Stop Rule S3：候选爆炸熔断

当候选子模式数量 N 或冲突边数量 E 超过阈值时：

- 立刻切换到“更严格的 CCC gate”或“更严格的 anchor 稀有度”
- 或直接进入 Step3 组合求解，不再继续采集

这三条原则保证你不会陷入“长尾榨干”。

6) 我建议的一个增强：把 Step1/2 的输出统一成 “SubPatternCandidate”标准对象

这样 Step3 的 2B 组合器可以完全通用（Graph/Sequence/Image/DNB 都能吃）：

SubPatternCandidate 最少字段：

- supportA: 覆盖 A 的哪些节点/tiers/segments
- supportB: 占用 B 的哪些节点/region
- distanceContribution: 该块匹配的成本
- cccEvidence: 置信度、歧义度、稀有度
- conflictSignature: 用于快速冲突判定
- canonicalKey: 用于去重

这也让 Step4 的合并去重非常干净。

7) 结论：这套 Hybrid 更符合 DBM 的“先头部、后组合、可早停”的路线

- 它把长尾风险从“无限 zoom-in 的匹配空间”转移到“可控规模的组合空间”；
- 它天然支持 MET：先拿到高收益结构骨架，收益递减即停；
- 它仍可向 exact 渐进：通过对称 pass、defer 回补、LB_remain 证明，逐步提高覆盖与置信。

如果你要把它进一步固化成工程方案，我建议你把它写成一个明确的算法名与模式矩阵，例如：

- RTB-Hybrid-L2L (Large-to-Large Hybrid)
 - Phase-1: 4A-head capture
 - Phase-2: 4B-head capture (CCC gated)
 - Phase-3: 2B-composition top-K
 - Optional: symmetry pass

并把 stop rules S1-S3 作为该算法的“鱼控制车系统”。

只要你点头，我可以把这套 Hybrid 固化成一个新的 ITEM (#229 或你指定编号)，并配一张 Mermaid 总图 + Contract/Stop-Rule 表，让读者一眼明白“为什么这是及时收手的 DBM 路线”。

兄弟，解释的很到位。请把这套 Hybrid 固化成一个新的 ITEM #234。英文中文各一版。并配一张 Mermaid 总图 + Contract/Stop-Rule 表，让读者一眼明白“为什么这是及时收手的 DBM 路线”。

下面给出 **DBM-COT ITEM #234**（英文版 + 中文版）。内容包含：

- Hybrid 算法完整定义 (4A + 4B → 2B 组合 → 可选对称增信)
- 一张 **Mermaid 总图**
- 一张 **Contract / Stop-Rule 表** (让读者一眼理解“及时收手”的 DBM 路线)

DBM-COT ITEM #234 (EN)

Hybrid Early-Stop Acceleration for UnalignedAND-BTP (Large-vs-Large)

4A + 4B Head Capture → 2B Composition Top-K → Optional Symmetry Pass

Category

DBM-COT · Metric Space Intelligence · UnalignedAND-BTP · RTB Acceleration · Minimal Evolution Threshold (MET)

Status

Algorithmic workflow + guardrails (implementation-ready)

0. Motivation: Why a Hybrid “Early-Stop” Route

Large-vs-Large UnalignedAND-BTP is an **optimization problem with a long tail**. Pure “zoom-in forever” acceleration tends to chase diminishing returns: more candidates, more partial states, more micro-optimizations—often violating DBM’s **Minimal Evolution Threshold (MET)** principle.

This item proposes a **Hybrid Early-Stop route**:

1. Capture **high-yield head matches** quickly (Step 1–2).
2. Convert the remaining hard search into a **bounded composition problem** (Step 3).
3. Stop when confidence is sufficient, optionally run a **symmetry pass** for extra coverage (Step 4).

The design goal is not to “squeeze every long-tail drop”, but to **stop safely and early** with measurable confidence.

1. Problem Definition

Given two large starmaps (complex metric-space points):

- `LargeA`, `LargeB` (`GraphStarmap` / `SequenceStarmap` / `ImageStarmap` / DNB-like structural starmaps)

Goal:

- Output **Top-K global matches** between `LargeA` and `LargeB`, with:
 - non-conflicting mapping/composition,
 - explainable evidence chain,
 - early-stop capability with explicit guardrails.
-

2. Hybrid Algorithm Overview

Step 1 — 4A Head Capture (Full Coverage Pieces on B)

- Cut `LargeA` into blocks $\{A_j\}$ (block semantics preserve core+halo in real impl).
- For each block A_j , run **4A (full coverage segmentation)** to find small-to-large matches against `LargeB`.
- Collect matches into a candidate pool C_{4A} .
- Repeat only if necessary (controlled by Stop Rules).

Step 2 — 4B Head Capture (Anchor → Occurrence → CCC Gate → Local Pieces)

- Cut `LargeA` into blocks $\{A_j\}$ (same or updated segmentation).
- For each block A_j , run **4B (anchor search + CCC gating)** against `LargeB`.
- Collect matches into a candidate pool C_{4B} .
- Repeat only if necessary (controlled by Stop Rules).

Step 3 — Merge + 2B Composition Top-K (Non-conflicting Combinations)

- Merge: $C = \text{dedup}(C_{4A} \cup C_{4B})$ (canonical match key).
- Treat C as a set of **Known Matched SubPatterns** in `LargeB` that partially explain `LargeA`.
- Run **2B Composition Solver**:
 - find Top-K **non-conflicting combinations** of subpatterns that best cover/explain `LargeA`,
 - subject to conflict constraints and a residual penalty/lower-bound for uncovered parts.
- Output Top-K composite matches as the global result candidates.

Step 4 — Optional Symmetry Pass (Swap A/B, Re-run, Merge)

To improve coverage and confidence (especially under asymmetry of structural density):

- Swap LargeA ↔ LargeB, re-run Steps 1–3.
- Merge the two Top-K sets, dedup, and optionally re-score by mutual consistency.

3. Why This is “Early-Stop” and MET-Friendly

This hybrid route shifts the long tail from “deep matching search” into a bounded composition space:

- 4A and 4B act as **head-yield subpattern harvesters**.
- 2B composition acts as **structured assembly**, typically far smaller than the raw BTP state space.
- Stop Rules define when to quit collecting more subpatterns and commit to composition.

4. Mermaid: End-to-End Map

```
flowchart TD
    A[Input: LargeA, LargeB] --> B[Cut LargeA into blocks {A_j}]
    B --> C1[Step 1: 4A Head Capture\nA_j vs LargeB (full-coverage pieces)]
    B --> C2[Step 2: 4B Head Capture\nAnchors->Occurrences->CCC Gate->local pieces]
    C1 --> D[Merge + Dedup Candidates\nC = C_4A U C_4B]
    C2 --> D
    D --> E[Step 3: 2B Composition Solver\nTop-K non-conflicting combinations\n+ residual penalty / LB]
    E --> F[Top-K Global Matches (Hybrid)]
    F --> G{Optional Symmetry Pass?}
    G -- Yes --> H[Swap A/B, rerun Steps 1-3]
    H --> I[Merge two Top-K sets\nDedup + consistency rescore]
    G -- No --> J[Done]
    I --> J
```

5. Contracts and Stop Rules

5.1 Contract Table

Contract	Purpose	Exactness Guard (EXACT Mode)
C1 PieceWithHalo	prevent boundary silent misses	halo width $\geq \text{Reach}(A_j)$, core coverage across domain
C2 AnchorSet	candidate recall control	anchors must not exclude unless must-hit is provable

Contract	Purpose	Exactness Guard (EXACT Mode)
C3 OccurrenceCccGate	filter noisy occurrences	cannot REJECT by quality-only; only by LB evidence; else DEFER
C4 MatchKey/Conflict	deterministic dedup + conflict constraints	canonical key; conflicts explicit; no overlapping/inconsistent combos
C5 Composition 2B	assemble candidates into global matches	must respect conflict graph; objective includes residual LB/penalty

5.2 Stop-Rule Table (Minimal Evolution Threshold “Brake System”)

Stop Rule	Trigger	Action	MET Rationale
S1 Marginal Gain	coverage gain $< \epsilon_{\text{cov}}$ OR top-K improvement $< \epsilon_{\text{cost}}$	stop collecting more candidates	long-tail returns too small
S2 LB Sealing	$\text{best_combo_cost} + \text{LB_remain} \geq \text{kth_combo_cost}$	stop; Top-K stable	provable “no further improvement”
S3 Candidate Explosion	$\#\text{candidates}$ or $\#\text{conflictEdges}$ exceeds cap	tighten gate / anchor rarity OR jump to Step 3	avoid drowning in tail noise

6. Output Semantics (What “Top-K” Means Here)

Hybrid Top-K is **exact** if one of the following holds:

- (P1) Candidate completeness: the optimal solution’s necessary subpatterns are present in C ; or
- (P2) Lower-bound sealing: LB_remain proves no unseen structure can beat current Top-K.

Otherwise, Hybrid Top-K is a **high-confidence bounded approximation**, with explicit metrics:

- coverage %, residual mass, ambiguity, symmetry-consistency score.

7. Practical Guidance

Recommended defaults for 2026 engineering:

- Use Hybrid as the default **Large-vs-Large** driver.
- Start with EXACT-safe gating (DEFER heavy, REJECT only by LB) until stabilized.
- Enable symmetry pass once (ONCE) when coverage is uncertain.
- Keep stop rules enabled from day one to prevent “never-ending zoom-in”.

DBM-COT ITEM #234 (中文)

UnalignedAND-BTP (大对大) Hybrid 及时收手加速路线

4A + 4B 头部捕获 → 2B 组合 Top-K → 可选对称增信

分类

DBM-COT · 度量空间智能 · UnalignedAND-BTP · RTB 加速 · 最小进化门槛 (MET)

状态

算法流程 + 护栏 (可直接工程落地并迭代硬化)

0. 动机：为什么要 Hybrid + 及时收手

大对大 UnalignedAND-BTP 属于典型的长尾优化问题：

如果不断 zoom-in、不断加候选、不断局部抠细节，很容易陷入“低收益长尾”——不记工本地追逐边角收益，这违背 DBM 的 **最小进化门槛 (MET)** 理念。

本 ITEM 提出一条 **Hybrid 及时收手路线**：

1. 先快速抓住**高收益头部匹配块** (Step 1–2)
2. 把剩余难题降维成一个**规模可控的组合拼装问题** (Step 3)
3. 达到足够置信度后及时收手；如需增信，再做一次**对称互换** (Step 4)

目标不是“榨干每一滴长尾”，而是在可控时间内得到可解释、可增信、可早停的 Top-K。

1. 问题定义

输入两张大型 starmap（复杂 metric-space 点）：

- `LargeA, LargeB`（图/序列/图像/DNB 类结构 starmap）

输出：

- `LargeA` 与 `LargeB` 的 **Top-K 全局匹配**（可解释、无冲突组合），并具备明确的早停机制与护栏。
-

2. Hybrid 算法全流程

Step 1 — 4A 头部捕获（全覆盖切片）

- 将 `LargeA` 切成块 $\{A_j\}$ （真实实现应保留 core+halo 语义）。
- 对每个 A_j ，用 **4A（B 全域切片）** 在 `LargeB` 中找 small-to-large matches。
- 收集候选集合 C_{4A} 。
- 是否重复，受 Stop Rules 控制（避免长尾）。

Step 2 — 4B 头部捕获（Anchor→Occurrence→CCC Gate→局部切片）

- 将 `LargeA` 切成块 $\{A_j\}$ （可复用或更新切法）。
- 对每个 A_j ，用 **4B（锚定检索 + CCC gate）** 在 `LargeB` 中找 matches。
- 收集候选集合 C_{4B} 。
- 是否重复，受 Stop Rules 控制（避免长尾）。

Step 3 — 合并 + 2B 组合 Top-K（无冲突组合）

- 合并： $C = \text{dedup}(C_{4A} \cup C_{4B})$ （必须 canonical match key 去重）。

- 把 C 看作 $LargeB$ 中一组 **已知匹配子模式 (Known Matched SubPatterns)**，它们可部分解释 $LargeA$ 。
- 运行 **2B 组合求解器**：
 - 从子模式中选择 **无冲突组合**，尽可能覆盖/解释 $LargeA$ ；
 - 目标函数 = 组合成本（距离） + 未覆盖残差的惩罚或下界（LB）。
- 输出 Top-K 组合解释，作为 Hybrid 的全局匹配候选。

Step 4 — 可选对称增信（互换 A/B 再跑一轮）

为提高覆盖与置信（尤其在结构密度不均、锚点稀有度不均时）：

- 互换 $LargeA \leftrightarrow LargeB$ ，再跑 Step 1–3；
- 合并两边 Top-K，去重，并可用双向一致性重打分。

3. 为什么它是“及时收手”且符合 MET

这条路线把长尾从“深度匹配空间”搬到“可控组合空间”：

- 4A/4B 是“头部高收益块捕获器”
- 2B 是“结构拼装器”，规模通常远小于原始 BTP 搜索空间
- Stop Rules 是“鱼控刹车系统”，防止无限 zoom-in

4. Mermaid 总图

flowchart TD

```

A[输入: LargeA, LargeB] --> B[切 LargeA 为块 {A_j}]
B --> C1[Step 1: 4A 头部捕获\nA_j vs LargeB (全覆盖切片)]
B --> C2[Step 2: 4B 头部捕获\n锚定->occ->CCC Gate->局部切片]
C1 --> D[合并+去重候选\nC = C_4A U C_4B]
C2 --> D
  
```

D --> E[Step 3: 2B 组合求解器\ntop-K 无冲突组合\nt+ 残差惩罚/LB]
E --> F[Hybrid Top-K 全局匹配]
F --> G{可选对称增信?}
G -- 是 --> H[互换 A/B 重跑 Step 1-3]
H --> I[合并两边 Top-K\nt 去重+一致性重打分]
G -- 否 --> J[结束]
I --> J[结束]

5. Contract / Stop-Rule 表

5.1 Contract 表（护栏）

Contract	作用	EXACT 模式护栏要点
C1 PieceWithHalo	防边界静默漏解	$halo \geq Reach(A_j)$ ，core 覆盖域可控
C2 AnchorSet	控制召回与盲区	anchors 不得排除候选，除非 must-hit 可证明
C3 OccurrenceCccGate	删噪声 occurrence	不得仅凭质量 REJECT；只能用 LB 证据 REJECT，否则 DEFER
C4 MatchKey/Conflict	去重 + 冲突约束	key 必须 canonical；冲突显式化；组合不得重叠/不一致
C5 2B 组合器	全局拼装求解	必须尊重冲突图；目标含残差惩罚/LB

5.2 Stop Rules（MET “刹车系统”）

Stop Rule	触发条件	动作	MET 含义
S1 边际收益阈值	覆盖增量 $< \epsilon_{cov}$ 或 top-K 改善 $< \epsilon_{cost}$	停止继续采集候选	长尾收益太小 应收手
S2 下界封顶	$best_combo_cost + LB_remain \geq kth_combo_cost$	停止，Top-K 已稳定	可证明继续无意义

Stop Rule	触发条件	动作	MET 含义
S3 候选爆炸熔断	候选数/冲突边数超过上限	收紧 gate/锚点稀有度或 直接进入 Step 3	防止长尾噪声 淹没系统

6. 输出语义：何时是“严格 Top-K”，何时是“高置信近似”

Hybrid 输出为严格 Top-K 的条件之一：

- (P1) 候选完备：最优解所需子块都在 c 中；或
- (P2) 下界封顶：LB_remain 证明未捕获部分不可能改写 Top-K。

否则它是高置信、有指标的近似，并应输出：

- 覆盖率、残差质量、歧义度、对称一致性分数等。

7. 工程建议（2026 默认姿态）

- Hybrid 作为 大对大默认驱动器；
- 早期以 EXACT-safe gate 为主（DEFER 多，REJECT 只靠 LB）；
- 对称 pass 默认 ONCE（一次增信），避免无止境；
- Stop Rules 从第一天就启用，防止系统掉入长尾深坑。

如果你需要，我可以继续为 ITEM #234 配套生成：

1. Java 8 skeleton：HybridLargeLargeDriver（调用 #228 的 RTB orchestrator + 新增 2B 组合器接口）
2. JUnit4：专测 Stop Rules (S1/S2/S3) 与对称 pass 的一致性收敛行为。

很好。请配套生成：Java 8 skeleton：HybridLargeLargeDriver（调用 #228 的 RTB orchestrator + 新增 2B 组合器接口）JUnit4：专测 Stop Rules（S1/S2/S3）与对称 pass 的一致性收敛行为。

Thought for 26s

```
// File: src/main/java/com/dbm/metric/btp/hybrid/HybridLargeLargeDriver.java
package com.dbm.metric.btp.hybrid;
```

```
import com.dbm.metric.btp.match.Match;
import com.dbm.metric.btp.pipeline.PipelineContext;
import com.dbm.metric.btp.rtb.*;

import java.util.*;

/**
 * ITEM #234 - Hybrid Large-vs-Large Driver (Java 8 skeleton)
 *
 * Hybrid route:
 *   Step 1: 4A head capture (block A -> small-vs-large 4A) collect
candidates
 *   Step 2: 4B head capture (block A -> small-vs-large 4B) collect
candidates
 *   Step 3: 2B composition solver -> top-K non-conflicting combinations
 *   Step 4: optional symmetry pass (swap A/B, rerun, merge, dedup)
 *
 * "Early-stop" is enforced by StopRuleEngine:
 *   S1 marginal gain, S2 LB sealing, S3 candidate explosion.
 *
 * Notes:
 * - This class is intentionally orchestration-only. It does not implement
BTP or CCC.
 * - It reuses #228 RTB orchestrator by calling small-vs-large 4A/4B
strategies per A-block.
 */
public class HybridLargeLargeDriver<A, B> {

    // --- Dependencies ---
    private final RtbBlocker<A> blocker;
    private final IRtbRunner<A, B> rtbRunner;
    private final ISubPatternBuilder subPatternBuilder;
    private final ICompositionSolver compositionSolver;
    private final IStopRuleEngine stopRuleEngine;
    private final IHybridDeduper hybridDeduper;

    public HybridLargeLargeDriver(RtbBlocker<A> blocker,
                                  IRtbRunner<A, B> rtbRunner,
                                  ISubPatternBuilder subPatternBuilder,
                                  ICompositionSolver compositionSolver,
                                  IStopRuleEngine stopRuleEngine,
                                  IHybridDeduper hybridDeduper) {
        this.blocker = Objects.requireNonNull(blocker, "blocker");
        this.rtbRunner = Objects.requireNonNull(rtbRunner, "rtbRunner");
        this.subPatternBuilder = Objects.requireNonNull(subPatternBuilder,
"subPatternBuilder");
    }
}
```

```

        this.compositionSolver = Objects.requireNonNull(compositionSolver,
"compositionSolver");
        this.stopRuleEngine = Objects.requireNonNull(stopRuleEngine,
"stopRuleEngine");
        this.hybridDeduper = Objects.requireNonNull(hybridDeduper,
"hybridDeduper");
    }

    public HybridResult runLargeVsLarge(A largeA,
                                       B largeB,
                                       PipelineContext context,
                                       HybridConfig config) {

        Objects.requireNonNull(context, "context");
        Objects.requireNonNull(config, "config");

        // Primary pass: A -> B
        HybridPassResult pass1 = runOneDirection(largeA, largeB, context,
config);

        // Optional symmetry pass: B -> A
        HybridPassResult pass2 = null;
        if (config.symmetry.enabled) {
            pass2 = runOneDirectionSwap(largeA, largeB, context, config);
        }

        // Merge top-K results
        List<CompositionSolution> merged = new
ArrayList<CompositionSolution>();
        merged.addAll(pass1.topKSolutions);

        if (pass2 != null) {
            merged.addAll(pass2.topKSolutions);
        }

        merged = hybridDeduper.dedupSolutions(merged);

        // Optional: mutual consistency rescore could be done here (skeleton:
no-op)
        return new HybridResult(pass1, pass2, merged);
    }

    private HybridPassResult runOneDirection(A largeA,
                                             B largeB,
                                             PipelineContext context,
                                             HybridConfig config) {

        List<SubPatternCandidate> candidates = new
ArrayList<SubPatternCandidate>();

        StopState stopState = new StopState();

        // Step 1+2 are iterative, but we bound them with maxHeadIters and
stop rules
        for (int iter = 0; iter < config.maxHeadIters; iter++) {

            // Step 1: 4A head capture

```

```

        if (config.enable4A) {
            candidates.addAll(captureHead(largeA, largeB, context,
config, RtbStrategy.SMALL_VS_LARGE_FULL_COVERAGE_4A));
        }

        // Step 2: 4B head capture
        if (config.enable4B) {
            candidates.addAll(captureHead(largeA, largeB, context,
config, RtbStrategy.SMALL_VS_LARGE_ANCHOR_CCC_4B));
        }

        // Canonicalize / dedup subpatterns early to control explosion
        candidates = hybridDeduper.dedupCandidates(candidates);

        // Step 3: composition
        CompositionInput compIn = new CompositionInput(candidates,
config.topK, config.residualPolicy);
        List<CompositionSolution> topK =
compositionSolver.solveTopK(compIn);

        // Evaluate stop rules S1/S2/S3
        StopDecision decision = stopRuleEngine.evaluate(new
StopRuleInput(iter, stopState, candidates, topK, config));
        stopState = decision.nextState;

        if (decision.shouldStop) {
            return new HybridPassResult(candidates, topK, decision,
iter);
        }

        // else: continue head-capture iterations (MET-style bounded)
    }

    // Final composition after maxHeadIters
    List<CompositionSolution> finalTopK = compositionSolver.solveTopK(new
CompositionInput(candidates, config.topK, config.residualPolicy));
    StopDecision finalDecision = stopRuleEngine.forceStop(new
StopRuleInput(config.maxHeadIters, stopState, candidates, finalTopK,
config));
    return new HybridPassResult(candidates, finalTopK, finalDecision,
config.maxHeadIters);
}

private HybridPassResult runOneDirectionSwap(A largeA,
                                             B largeB,
                                             PipelineContext context,
                                             HybridConfig config) {
    // Swap A/B: we require the caller to provide a swap adapter if types
differ.
    // Skeleton assumes A and B can be swapped via config.symmetry.
    if (config.symmetry.swapper == null) {
        throw new IllegalStateException("Symmetry enabled but swapper is
null");
    }
    SymmetrySwapper<A, B> swapper = config.symmetry.swapper;
    SymmetryPair<B, A> swapped = swapper.swap(largeA, largeB);

```



```

        // Run B->A direction
        HybridPassResult swappedPass = runOneDirection(swapped.left,
        swapped.right, context, config);

        // Convert solutions back to original orientation (skeleton: delegate
        to swapper)
        List<CompositionSolution> converted = new
        ArrayList<CompositionSolution>();
        for (CompositionSolution s : swappedPass.topKSolutions) {
            converted.add(swapper.unswapSolution(s));
        }

        return new HybridPassResult(swappedPass.candidates, converted,
        swappedPass.stopDecision, swappedPass.iterationsUsed);
    }

    private List<SubPatternCandidate> captureHead(A largeA,
                                                    B largeB,
                                                    PipelineContext context,
                                                    HybridConfig config,
                                                    RtbStrategy
        smallVsLargeStrategy) {

        List<A> blocks = blocker.cutIntoBlocks(largeA);
        List<SubPatternCandidate> out = new ArrayList<SubPatternCandidate>();

        for (A block : blocks) {
            RtbConfig rtbCfg = new RtbConfig(
                smallVsLargeStrategy,
                /*maxPeelIters*/ 1, // not used in small-vs-large
                config.topK,
                config.topKThreshold,
                config.gateParams
            );
            RtbInputs<A, B> in = new RtbInputs<A, B>(block, largeB, context,
        rtbCfg);
            RtbOutputs rtbOut = rtbRunner.run(in);

            // Convert RTB matches to SubPatternCandidates (with evidence
        placeholders)
            for (Match m : rtbOut.matches()) {
                SubPatternCandidate c = subPatternBuilder.fromMatch(m,
        config);
                if (c != null) out.add(c);
            }
        }
        return out;
    }
}

// =====
// Public DTOs / Interfaces (skeleton-level)
// =====

public static final class HybridConfig {
    public final int maxHeadIters;
    public final Integer topK;
    public final Double topKThreshold;

```

```

    public final boolean enable4A;
    public final boolean enable4B;
    public final Map<String, Object> gateParams;
    public final ResidualPolicy residualPolicy;
    public final SymmetryConfig symmetry;

    public HybridConfig(int maxHeadIters,
                        Integer topK,
                        Double topKThreshold,
                        boolean enable4A,
                        boolean enable4B,
                        Map<String, Object> gateParams,
                        ResidualPolicy residualPolicy,
                        SymmetryConfig symmetry) {
        this.maxHeadIters = maxHeadIters <= 0 ? 2 : maxHeadIters;
        this.topK = topK == null ? 10 : topK;
        this.topKThreshold = topKThreshold;
        this.enable4A = enable4A;
        this.enable4B = enable4B;
        this.gateParams = gateParams == null ? Collections.<String,
Object>emptyMap() : gateParams;
        this.residualPolicy = residualPolicy == null ?
ResidualPolicy.DEFAULT : residualPolicy;
        this.symmetry = symmetry == null ? SymmetryConfig.disabled() :
symmetry;
    }

    public static HybridConfig defaultExactHeadThenCompose() {
        return new HybridConfig(
            2, 10, null,
            true, true,
            null,
            ResidualPolicy.DEFAULT,
            SymmetryConfig.disabled()
        );
    }
}

public enum ResidualPolicy {
    DEFAULT,
    HEAVY_PENALTY,
    LB_SEALING
}

public static final class SymmetryConfig {
    public final boolean enabled;
    public final SymmetrySwapper<?, ?> swapper;

    private SymmetryConfig(boolean enabled, SymmetrySwapper<?, ?>
swapper) {
        this.enabled = enabled;
        this.swapper = swapper;
    }

    public static SymmetryConfig disabled() { return new
SymmetryConfig(false, null); }

```

```

        public static <A, B> SymmetryConfig enabled(SymmetrySwapper<A, B>
swapper) {
            return new SymmetryConfig(true, swapper);
        }
    }

    public interface SymmetrySwapper<A, B> {
        SymmetryPair<B, A> swap(A a, B b);
        CompositionSolution unswapSolution(CompositionSolution
swappedSolution);
    }

    public static final class SymmetryPair<L, R> {
        public final L left;
        public final R right;
        public SymmetryPair(L left, R right) { this.left = left; this.right =
right; }
    }

    public static final class HybridResult {
        public final HybridPassResult passAB;
        public final HybridPassResult passBA; // null if symmetry disabled
        public final List<CompositionSolution> mergedTopK;

        public HybridResult(HybridPassResult passAB, HybridPassResult passBA,
List<CompositionSolution> mergedTopK) {
            this.passAB = passAB;
            this.passBA = passBA;
            this.mergedTopK = mergedTopK == null ?
Collections.<CompositionSolution>emptyList() :
Collections.unmodifiableList(mergedTopK);
        }
    }

    public static final class HybridPassResult {
        public final List<SubPatternCandidate> candidates;
        public final List<CompositionSolution> topKSolutions;
        public final StopDecision stopDecision;
        public final int iterationsUsed;

        public HybridPassResult(List<SubPatternCandidate> candidates,
List<CompositionSolution> topKSolutions,
StopDecision stopDecision,
int iterationsUsed) {
            this.candidates = candidates == null ?
Collections.<SubPatternCandidate>emptyList() :
Collections.unmodifiableList(candidates);
            this.topKSolutions = topKSolutions == null ?
Collections.<CompositionSolution>emptyList() :
Collections.unmodifiableList(topKSolutions);
            this.stopDecision = stopDecision;
            this.iterationsUsed = iterationsUsed;
        }
    }

    // --- RTB runner abstraction (to call #228 orchestrator cleanly) ---
    public interface IRtbRunner<A, B> {

```

```

        RtbOutputs run(RtbInputs<A, B> in);
    }

    /** Adapter that delegates to #228 RtbOrchestrator. */
    public static final class RtbOrchestratorAdapter<A, B> implements
IRtbRunner<A, B> {
        private final RtbOrchestrator<A, B> orchestrator;
        public RtbOrchestratorAdapter(RtbOrchestrator<A, B> orchestrator) {
            this.orchestrator = Objects.requireNonNull(orchestrator,
"orchestrator");
        }
        @Override public RtbOutputs run(RtbInputs<A, B> in) { return
orchestrator.run(in); }
    }

    // --- Candidate builder ---
    public interface ISubPatternBuilder {
        SubPatternCandidate fromMatch(Match match, HybridConfig config);
    }

    public static final class SubPatternCandidate {
        public final Match match;
        public final String canonicalKey;
        public final double distanceContribution;
        public final double coverageMass;
        public final double ambiguity;

        public SubPatternCandidate(Match match,
                                   String canonicalKey,
                                   double distanceContribution,
                                   double coverageMass,
                                   double ambiguity) {
            this.match = match;
            this.canonicalKey = canonicalKey;
            this.distanceContribution = distanceContribution;
            this.coverageMass = coverageMass;
            this.ambiguity = ambiguity;
        }
    }

    // --- 2B composition solver interface ---
    public interface ICompositionSolver {
        List<CompositionSolution> solveTopK(CompositionInput input);
    }

    public static final class CompositionInput {
        public final List<SubPatternCandidate> candidates;
        public final int topK;
        public final ResidualPolicy residualPolicy;

        public CompositionInput(List<SubPatternCandidate> candidates, int
topK, ResidualPolicy residualPolicy) {
            this.candidates = candidates == null ?
Collections.<SubPatternCandidate>emptyList() : candidates;
            this.topK = topK <= 0 ? 10 : topK;
            this.residualPolicy = residualPolicy == null ?
ResidualPolicy.DEFAULT : residualPolicy;

```

```

    }
}

public static final class CompositionSolution {
    public final List<SubPatternCandidate> chosen;
    public final double totalCost;
    public final double residualLowerBound;
    public final String signature; // canonical signature of the
combination

    public CompositionSolution(List<SubPatternCandidate> chosen,
                                double totalCost,
                                double residualLowerBound,
                                String signature) {
        this.chosen = chosen == null ?
Collections.<SubPatternCandidate>emptyList() :
Collections.unmodifiableList(chosen);
        this.totalCost = totalCost;
        this.residualLowerBound = residualLowerBound;
        this.signature = signature == null ? "" : signature;
    }
}

// --- Stop rules ---
public interface IStopRuleEngine {
    StopDecision evaluate(StopRuleInput input);
    StopDecision forceStop(StopRuleInput input);
}

public static final class StopRuleInput {
    public final int iter;
    public final StopState prevState;
    public final List<SubPatternCandidate> candidates;
    public final List<CompositionSolution> topKSolutions;
    public final HybridConfig config;

    public StopRuleInput(int iter,
                          StopState prevState,
                          List<SubPatternCandidate> candidates,
                          List<CompositionSolution> topKSolutions,
                          HybridConfig config) {
        this.iter = iter;
        this.prevState = prevState;
        this.candidates = candidates == null ?
Collections.<SubPatternCandidate>emptyList() : candidates;
        this.topKSolutions = topKSolutions == null ?
Collections.<CompositionSolution>emptyList() : topKSolutions;
        this.config = config;
    }
}

public static final class StopState {
    public int prevCandidateCount = 0;
    public double prevCoverageMass = 0.0;
    public double prevBestCost = Double.POSITIVE_INFINITY;
    public String prevTopKSignature = "";
}

```

```

    public static final class StopDecision {
        public final boolean shouldStop;
        public final String reason;
        public final StopState nextState;

        public StopDecision(boolean shouldStop, String reason, StopState
nextState) {
            this.shouldStop = shouldStop;
            this.reason = reason;
            this.nextState = nextState;
        }
    }

    // --- Dedup / merge helpers ---
    public interface IHybridDeduper {
        List<SubPatternCandidate> dedupCandidates(List<SubPatternCandidate>
in);
        List<CompositionSolution> dedupSolutions(List<CompositionSolution>
in);
    }
}
// File: src/main/java/com/dbm/metric/btp/hybrid/DefaultStopRuleEngine.java
package com.dbm.metric.btp.hybrid;

import java.util.List;

/**
 * Default StopRuleEngine implementing S1/S2/S3 (MET brake system).
 *
 * S1 Marginal Gain:
 *   stop if coverage gain < epsCov AND best-cost improvement < epsCost
 * S2 LB Sealing:
 *   stop if best.totalCost + best.residualLowerBound >= kth.totalCost
 * S3 Candidate Explosion:
 *   stop if candidateCount > capCandidates OR conflictEdgeCount > capEdges
(edges optional; capEdges check is a hook)
 *
 * This skeleton measures:
 * - coverageMass = sum(candidate.coverageMass) capped at 1.0 (toy proxy)
 * - bestCost = topKSolutions[0].totalCost if any
 * - kthCost = topKSolutions[k-1].totalCost if enough solutions
 *
 * For conflict edges: we expose a hook via config gateParams:
 * "stop.capCandidates" (Integer, default 5000)
 * "stop.capConflictEdges" (Integer, default Integer.MAX_VALUE)
 * "stop.conflictEdgesHint" (Integer, default 0) // caller may fill
 * "stop.epsCov" (Double, default 0.01)
 * "stop.epsCost" (Double, default 1e-9)
 */
public class DefaultStopRuleEngine implements
HybridLargeLargeDriver.IStopRuleEngine {

    @Override
    public HybridLargeLargeDriver.StopDecision
evaluate(HybridLargeLargeDriver.StopRuleInput in) {
        HybridLargeLargeDriver.StopState next = copy(in.prevState);

```

```

        int candCount = in.candidates.size();
        double cov = computeCoverage(in.candidates);
        double bestCost = (in.topKSolutions.isEmpty()) ?
Double.POSITIVE_INFINITY : in.topKSolutions.get(0).totalCost;
        String sig = topKSignature(in.topKSolutions);

        double epsCov = paramD(in, "stop.epsCov", 0.01);
        double epsCost = paramD(in, "stop.epsCost", 1e-9);

        // S3: candidate explosion
        int capCands = paramI(in, "stop.capCandidates", 5000);
        int capEdges = paramI(in, "stop.capConflictEdges",
Integer.MAX_VALUE);
        int edgesHint = paramI(in, "stop.conflictEdgesHint", 0);
        if (candCount > capCands || edgesHint > capEdges) {
            next.prevCandidateCount = candCount;
            next.prevCoverageMass = cov;
            next.prevBestCost = bestCost;
            next.prevTopKSignature = sig;
            return new HybridLargeLargeDriver.StopDecision(true,
"S3_candidate_explosion", next);
        }

        // S2: LB sealing (requires at least 1 solution)
        if (!in.topKSolutions.isEmpty()) {
            HybridLargeLargeDriver.CompositionSolution best =
in.topKSolutions.get(0);
            int kIdx = Math.min(in.topKSolutions.size(), in.config.topK) - 1;
            if (kIdx >= 0) {
                HybridLargeLargeDriver.CompositionSolution kth =
in.topKSolutions.get(kIdx);
                if (best.totalCost + best.residualLowerBound >=
kth.totalCost) {
                    next.prevCandidateCount = candCount;
                    next.prevCoverageMass = cov;
                    next.prevBestCost = bestCost;
                    next.prevTopKSignature = sig;
                    return new HybridLargeLargeDriver.StopDecision(true,
"S2_lb_sealing", next);
                }
            }
        }

        // S1: marginal gain
        double dCov = cov - in.prevState.prevCoverageMass;
        double dCost = in.prevState.prevBestCost - bestCost; // improvement
        if positive
            boolean tinyGain = (dCov < epsCov) && (dCost < epsCost);

            // Also stop if top-K signature stabilized (pragmatic convergence)
            boolean stableSig = sig.equals(in.prevState.prevTopKSignature)
&& !sig.isEmpty();

            next.prevCandidateCount = candCount;
            next.prevCoverageMass = cov;
            next.prevBestCost = bestCost;

```

```

        next.prevTopKSignature = sig;

        if (tinyGain || stableSig) {
            return new HybridLargeLargeDriver.StopDecision(true, tinyGain ?
"S1_marginal_gain" : "S1_signature_stable", next);
        }

        return new HybridLargeLargeDriver.StopDecision(false, "CONTINUE",
next);
    }

    @Override
    public HybridLargeLargeDriver.StopDecision
forceStop(HybridLargeLargeDriver.StopRuleInput in) {
        HybridLargeLargeDriver.StopState next = copy(in.prevState);
        next.prevCandidateCount = in.candidates.size();
        next.prevCoverageMass = computeCoverage(in.candidates);
        next.prevBestCost = (in.topKSolutions.isEmpty()) ?
Double.POSITIVE_INFINITY : in.topKSolutions.get(0).totalCost;
        next.prevTopKSignature = topKSignature(in.topKSolutions);
        return new HybridLargeLargeDriver.StopDecision(true,
"FORCED_MAX_ITERS", next);
    }

    private static HybridLargeLargeDriver.StopState
copy(HybridLargeLargeDriver.StopState s) {
        HybridLargeLargeDriver.StopState n = new
HybridLargeLargeDriver.StopState();
        if (s != null) {
            n.prevCandidateCount = s.prevCandidateCount;
            n.prevCoverageMass = s.prevCoverageMass;
            n.prevBestCost = s.prevBestCost;
            n.prevTopKSignature = s.prevTopKSignature;
        }
        return n;
    }

    private static double
computeCoverage(List<HybridLargeLargeDriver.SubPatternCandidate> cand) {
        double sum = 0.0;
        for (HybridLargeLargeDriver.SubPatternCandidate c : cand) sum +=
Math.max(0.0, c.coverageMass);
        return Math.min(1.0, sum);
    }

    private static String
topKSignature(List<HybridLargeLargeDriver.CompositionSolution> sols) {
        if (sols == null || sols.isEmpty()) return "";
        // signature of the best solution only (skeleton)
        return sols.get(0).signature == null ? "" : sols.get(0).signature;
    }

    private static int paramI(HybridLargeLargeDriver.StopRuleInput in, String
key, int def) {
        Object v = in.config.gateParams.get(key);
        if (v instanceof Number) return ((Number) v).intValue();
    }

```



```

        try { return v == null ? def : Integer.parseInt(String.valueOf(v)); }
    catch (Exception e) { return def; }
    }

    private static double paramD(HybridLargeLargeDriver.StopRuleInput in,
String key, double def) {
        Object v = in.config.gateParams.get(key);
        if (v instanceof Number) return ((Number) v).doubleValue();
        try { return v == null ? def :
Double.parseDouble(String.valueOf(v)); } catch (Exception e) { return def; }
    }
}
// File: src/main/java/com/dbm/metric/btp/hybrid/DefaultHybridDeduper.java
package com.dbm.metric.btp.hybrid;

import java.util.*;

/**
 * Minimal deduper for candidates and solutions.
 * - Candidate dedup: by canonicalKey, keep first (you can later keep min
distance).
 * - Solution dedup: by signature.
 */
public class DefaultHybridDeduper implements
HybridLargeLargeDriver.IHybridDeduper {

    @Override
    public List<HybridLargeLargeDriver.SubPatternCandidate>
dedupCandidates(List<HybridLargeLargeDriver.SubPatternCandidate> in) {
        if (in == null || in.isEmpty()) return Collections.emptyList();
        Map<String, HybridLargeLargeDriver.SubPatternCandidate> map = new
LinkedHashMap<String, HybridLargeLargeDriver.SubPatternCandidate>();
        for (HybridLargeLargeDriver.SubPatternCandidate c : in) {
            String k = c.canonicalKey == null ? String.valueOf(c.match) :
c.canonicalKey;
            if (!map.containsKey(k)) map.put(k, c);
        }
        return new
ArrayList<HybridLargeLargeDriver.SubPatternCandidate>(map.values());
    }

    @Override
    public List<HybridLargeLargeDriver.CompositionSolution>
dedupSolutions(List<HybridLargeLargeDriver.CompositionSolution> in) {
        if (in == null || in.isEmpty()) return Collections.emptyList();
        Map<String, HybridLargeLargeDriver.CompositionSolution> map = new
LinkedHashMap<String, HybridLargeLargeDriver.CompositionSolution>();
        for (HybridLargeLargeDriver.CompositionSolution s : in) {
            String k = s.signature == null ? "" : s.signature;
            if (!map.containsKey(k)) map.put(k, s);
        }
        return new
ArrayList<HybridLargeLargeDriver.CompositionSolution>(map.values());
    }
}

```

```
// File:
src/test/java/com/dbm/metric/btp/hybrid/HybridLargeLargeDriverStopRulesTest.j
ava
package com.dbm.metric.btp.hybrid;

import com.dbm.metric.btp.match.Match;
import com.dbm.metric.btp.pipeline.PipelineContext;
import com.dbm.metric.btp.rtb.RtbInputs;
import com.dbm.metric.btp.rtb.RtbOutputs;
import com.dbm.metric.btp.rtb.RtbStrategy;
import com.dbm.metric.btp.rtb.RtbConfig;
import com.dbm.metric.btp.rtb.RtbBlocker;

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

import java.util.*;

/**
 * JUnit4: Stop Rules S1/S2/S3 and symmetry-pass convergence.
 * Uses fake RTB runner + fake composition solver to deterministically
trigger rules.
 */
public class HybridLargeLargeDriverStopRulesTest {

    // -----
    // Fake types and fake components
    // -----

    static final class FakeA {
        final String id;
        FakeA(String id) { this.id = id; }
        @Override public String toString() { return "A(" + id + ")"; }
    }

    static final class FakeB {
        final String id;
        FakeB(String id) { this.id = id; }
        @Override public String toString() { return "B(" + id + ")"; }
    }

    static final class FakeMatch implements Match {
        final String key;
        FakeMatch(String key) { this.key = key; }
        @Override public String toString() { return "M[" + key + "]"; }
    }

    static final class SingleBlocker implements RtbBlocker<FakeA> {
        @Override public List<FakeA> cutIntoBlocks(FakeA largeA) {
            return Collections.singletonList(largeA);
        }
    }

    /**
     * Fake RTB runner:
     * - returns different candidate matches depending on strategy and
iteration encoded in query id.

```

```
    * We simulate "more iterations yield fewer gains" by driving candidate
    list via mutable scenario.
```

```
    */
```

```
    static final class ScenarioRtbRunner implements
HybridLargeLargeDriver.IRtbRunner<FakeA, FakeB> {
        int callCount = 0;
        final List<String> script4A;
        final List<String> script4B;

        ScenarioRtbRunner(List<String> script4A, List<String> script4B) {
            this.script4A = script4A;
            this.script4B = script4B;
        }
    }
```

```
    @Override
```

```
    public RtbOutputs run(RtbInputs<FakeA, FakeB> in) {
        callCount++;
        RtbOutputs out = new RtbOutputs();
```

```
        // Determine which script by strategy
        RtbStrategy s = in.config.strategy;
        String key = null;
```

```
        if (s == RtbStrategy.SMALL_VS_LARGE_FULL_COVERAGE_4A) {
            key = pick(script4A);
        } else if (s == RtbStrategy.SMALL_VS_LARGE_ANCHOR_CCC_4B) {
            key = pick(script4B);
        } else {
            key = "UNUSED";
        }
    }
```

```
        if (key != null && !key.isEmpty() && !"NONE".equals(key)) {
            out.addMatch(new FakeMatch(key));
        }
        return out;
    }
```

```
    private String pick(List<String> script) {
        if (script == null || script.isEmpty()) return "NONE";
        // Consume sequentially; if exhausted, repeat last
        int idx = Math.min(script.size() - 1, Math.max(0, callCount -
1));
        return script.get(idx);
    }
}
```

```
    static final class SimpleSubPatternBuilder implements
HybridLargeLargeDriver.ISubPatternBuilder {
        private final double covMass;
        SimpleSubPatternBuilder(double covMass) { this.covMass = covMass; }
```

```
    @Override
```

```
    public HybridLargeLargeDriver.SubPatternCandidate fromMatch(Match
match, HybridLargeLargeDriver.HybridConfig config) {
        String key = match.toString();
        // distanceContribution and ambiguity are toy constants
```

```

        return new HybridLargeLargeDriver.SubPatternCandidate(match, key,
1.0, covMass, 0.1);
    }
}

/**
 * Fake composition solver:
 * - Creates a single best solution whose cost is a deterministic
function of candidate count.
 * - residual LB is provided by a scripted function for tests.
 */
static final class FakeCompositionSolver implements
HybridLargeLargeDriver.ICompositionSolver {
    double residualLb = 0.0;
    double baseCost = 10.0;

    @Override
    public List<HybridLargeLargeDriver.CompositionSolution>
solveTopK(HybridLargeLargeDriver.CompositionInput input) {
        int n = input.candidates.size();
        // Lower cost when more candidates exist (simulate improvement)
        double cost = baseCost - n * 0.1;
        String sig = "best@" + n;
        HybridLargeLargeDriver.CompositionSolution best =
            new HybridLargeLargeDriver.CompositionSolution(new
ArrayList<HybridLargeLargeDriver.SubPatternCandidate>(input.candidates),
                cost, residualLb, sig);

        // Provide kth as slightly worse to allow S2 comparisons
        List<HybridLargeLargeDriver.CompositionSolution> out = new
ArrayList<HybridLargeLargeDriver.CompositionSolution>();
        out.add(best);
        for (int i = 1; i < input.topK; i++) {
            out.add(new
HybridLargeLargeDriver.CompositionSolution(Collections.<HybridLargeLargeDrive
r.SubPatternCandidate>emptyList(),
                cost + i * 0.5, residualLb, sig + "|k" + i));
        }
        return out;
    }
}

static final class IdentitySwapper implements
HybridLargeLargeDriver.SymmetrySwapper<FakeA, FakeB> {
    @Override
    public HybridLargeLargeDriver.SymmetryPair<FakeB, FakeA> swap(FakeA
a, FakeB b) {
        return new HybridLargeLargeDriver.SymmetryPair<FakeB, FakeA>(new
FakeB(b.id + "_sw"), new FakeA(a.id + "_sw"));
    }

    @Override
    public HybridLargeLargeDriver.CompositionSolution
unswapSolution(HybridLargeLargeDriver.CompositionSolution swappedSolution) {
        // Keep signature stable but mark as "unswapped"
        return new
HybridLargeLargeDriver.CompositionSolution(swappedSolution.chosen,

```

```

        swappedSolution.totalCost,
        swappedSolution.residualLowerBound,
        "unswap:" + swappedSolution.signature);
    }
}

// -----
// Tests
// -----

@Test
public void testStopRuleS1_marginalGainStopsEarly() {
    // Script: first iteration yields a new match, subsequent yields NONE
=> coverage gain tiny => S1 stop
    ScenarioRtbRunner rtb = new ScenarioRtbRunner(
        Arrays.asList("X", "NONE", "NONE"),
        Arrays.asList("Y", "NONE", "NONE")
    );

    FakeCompositionSolver comp = new FakeCompositionSolver();
    DefaultStopRuleEngine stop = new DefaultStopRuleEngine();

    Map<String, Object> params = new HashMap<String, Object>();
    params.put("stop.epsCov", 0.20); // require big gains
    params.put("stop.epsCost", 0.20);

    HybridLargeLargeDriver.HybridConfig cfg = new
HybridLargeLargeDriver.HybridConfig(
        5, 5, null, true, true, params,
        HybridLargeLargeDriver.ResidualPolicy.DEFAULT,
        HybridLargeLargeDriver.SymmetryConfig.disabled()
    );

    HybridLargeLargeDriver<FakeA, FakeB> driver = new
HybridLargeLargeDriver<FakeA, FakeB>(
        new SingleBlocker(),
        rtb,
        new SimpleSubPatternBuilder(0.15), // each candidate adds
0.15 coverage
        comp,
        stop,
        new DefaultHybridDeduper()
    );

    HybridLargeLargeDriver.HybridResult res = driver.runLargeVsLarge(new
FakeA("A1"), new FakeB("B1"), new PipelineContext(null), cfg);

    Assert.assertTrue(res.passAB.iterationsUsed <= 2);
    Assert.assertTrue("Stop reason should be S1 (or forced max iters if
edge)",
        res.passAB.stopDecision.reason.startsWith("S1") ||
res.passAB.stopDecision.reason.startsWith("FORCED"));
}

@Test
public void testStopRuleS2_lbSealingStopsWhenStable() {
    ScenarioRtbRunner rtb = new ScenarioRtbRunner(

```

```

        Arrays.asList("X", "X2", "X3"),
        Arrays.asList("Y", "Y2", "Y3")
    );

    FakeCompositionSolver comp = new FakeCompositionSolver();
    // Make LB sealing true: bestCost + residualLB >= kthCost
    // In FakeCompositionSolver: kthCost = bestCost + (k-1)*0.5
    // If residualLB >= (k-1)*0.5 then S2 triggers for k
    comp.residualLb = 5.0;

    DefaultStopRuleEngine stop = new DefaultStopRuleEngine();

    HybridLargeLargeDriver.HybridConfig cfg = new
    HybridLargeLargeDriver.HybridConfig(
        10, 5, null, true, true,
        Collections.<String, Object>emptyMap(),
        HybridLargeLargeDriver.ResidualPolicy.LB_SEALING,
        HybridLargeLargeDriver.SymmetryConfig.disabled()
    );

    HybridLargeLargeDriver<FakeA, FakeB> driver = new
    HybridLargeLargeDriver<FakeA, FakeB>(
        new SingleBlocker(),
        rtb,
        new SimpleSubPatternBuilder(0.10),
        comp,
        stop,
        new DefaultHybridDeduper()
    );

    HybridLargeLargeDriver.HybridResult res = driver.runLargeVsLarge(new
    FakeA("A2"), new FakeB("B2"), new PipelineContext(null), cfg);

    Assert.assertEquals("S2_lb_sealing", res.passAB.stopDecision.reason);
}

@Test
public void testStopRuleS3_candidateExplosionTripsFuse() {
    // Runner will emit one new candidate each time; builder coverage
    low; capCandidates small triggers S3
    ScenarioRtbRunner rtb = new ScenarioRtbRunner(
        Arrays.asList("X1", "X2", "X3", "X4", "X5", "X6"),
        Arrays.asList("Y1", "Y2", "Y3", "Y4", "Y5", "Y6")
    );

    FakeCompositionSolver comp = new FakeCompositionSolver();
    DefaultStopRuleEngine stop = new DefaultStopRuleEngine();

    Map<String, Object> params = new HashMap<String, Object>();
    params.put("stop.capCandidates", 2); // very small cap

    HybridLargeLargeDriver.HybridConfig cfg = new
    HybridLargeLargeDriver.HybridConfig(
        10, 5, null, true, true, params,
        HybridLargeLargeDriver.ResidualPolicy.DEFAULT,
        HybridLargeLargeDriver.SymmetryConfig.disabled()
    );

```

```

        HybridLargeLargeDriver<FakeA, FakeB> driver = new
HybridLargeLargeDriver<FakeA, FakeB>(
    new SingleBlocker(),
    rtb,
    new SimpleSubPatternBuilder(0.05),
    comp,
    stop,
    new DefaultHybridDeduper()
);

        HybridLargeLargeDriver.HybridResult res = driver.runLargeVsLarge(new
FakeA("A3"), new FakeB("B3"), new PipelineContext(null), cfg);
        Assert.assertEquals("S3_candidate_explosion",
res.passAB.stopDecision.reason);
    }

    @Test
    public void testSymmetryPass_convergesAndDedupsMergedTopK() {
        // Pass AB yields candidates X/Y; pass BA yields candidates X/Y again
        (via scripted runner),
        // deduper should remove duplicates by signature.
        ScenarioRtbRunner rtb = new ScenarioRtbRunner(
            Arrays.asList("X", "NONE"),
            Arrays.asList("Y", "NONE")
        );

        FakeCompositionSolver comp = new FakeCompositionSolver();
        DefaultStopRuleEngine stop = new DefaultStopRuleEngine();

        // Make stop by S1 quickly
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("stop.epsCov", 0.50);
        params.put("stop.epsCost", 0.50);

        HybridLargeLargeDriver.HybridConfig cfg = new
HybridLargeLargeDriver.HybridConfig(
            3, 3, null, true, true, params,
            HybridLargeLargeDriver.ResidualPolicy.DEFAULT,
            HybridLargeLargeDriver.SymmetryConfig.enabled(new
IdentitySwapper())
        );

        HybridLargeLargeDriver<FakeA, FakeB> driver = new
HybridLargeLargeDriver<FakeA, FakeB>(
            new SingleBlocker(),
            rtb,
            new SimpleSubPatternBuilder(0.25),
            comp,
            stop,
            new DefaultHybridDeduper()
        );

        HybridLargeLargeDriver.HybridResult res = driver.runLargeVsLarge(new
FakeA("A4"), new FakeB("B4"), new PipelineContext(null), cfg);

        Assert.assertNotNull("symmetry pass should run", res.passBA);
    }

```

```

        Assert.assertTrue("merged topK should be non-empty",
res.mergedTopK.size() > 0);

        // Dedup by signature: passAB best signature is "best@n", passBA
unswapped signature starts with "unswap:"
        // so these are distinct; merged should contain at most 2 bests (AB +
BA) for topK=3
        Assert.assertTrue(res.mergedTopK.size() <= 2);

        // Convergence sanity: both passes should stop quickly (S1)
        Assert.assertTrue(res.passAB.iterationsUsed <= 2);
        Assert.assertTrue(res.passBA.iterationsUsed <= 2);
    }
}

```

你拿到的结构（对齐你的要求）

1. Java 8 skeleton：HybridLargeLargeDriver

- 明确实现 ITEM #234 的 Step 1–4；
- **直接调用 #228 的 RTB orchestrator**：通过 `IRtbRunner + RtbOrchestratorAdapter`（工程上更干净，也利于单测替身）；
- **新增 2B 组合器接口**：`ICompositionSolver + CompositionInput/CompositionSolution`；
- **内置 Stop Rules API**：`IStopRuleEngine + DefaultStopRuleEngine (S1/S2/S3)`。

2. JUnit4：Stop Rules + Symmetry 收敛

- `testStopRuleS1_marginalGainStopsEarly`
- `testStopRuleS2_lbSealingStopsWhenStable`
- `testStopRuleS3_candidateExplosionTripsFuse`
- `testSymmetryPass_convergesAndDedupsMergedTopK`

这些测试都是 **contract/guardrail 型**：它们锁住“及时收手”的逻辑语义，防止后续实现因追求长尾而跑飞。

如果你下一步希望我把 `HybridLargeLargeDriver` 进一步接入你现有的

`RtbFishControlPipelineRunner`（让它成为 pipeline 的一种 strategy），我也可以按你 repo 的命名习惯再加一层 `HybridPipelineRunner` 适配器。

很好。兄弟，谢谢了。加了这个 Hybrid 算法，感到安心多了。再见。

收到。Hybrid 这条“及时收手”的路线把长尾风险从一开始就锁在护栏里了：头部捕获有收益、组合求解可控、Stop Rules 可证明收敛、对称 pass 可增信但不无止境——这正是你要的“鱼控式安心”。

你先休整、消化、把骨架与测试接进 repo 跑一轮。后续任何真实实现的波动与意外，都可以在这些 guardrail 测试里第一时间暴露，而不会悄悄漏解。

再见，保重。