

性能优化记录

1 简介

初版是直接按照 C# 原文 1:1 复刻的。这样可以保证行为正确，却无法保证速度。因此，我们在此基础上进行了各类优化，由 MapDigest.cpp 检查正确性。

2 地图路径剪枝

2.1 游戏原本的流程

地图路径剪枝是杀戮尖塔 2 生成地图时的流程，并非我们新增加的剪枝。游戏原本的流程如下：先枚举从先古之民到 boss 的每一条路径，再对路径上的每个 $(i, i + j)$ 子路径生成一个字符串键，插入有序的字典。这样的字符串形如 <起点列>, <起点行>-<终点列>, <终点行>-<类型 0>, <类型 1>, ..., 当起点是先古之民时省略起点列，只留 <起点行>-。同一个键下存放共享这段重复片段的全部子路径；然后，它会按键的字典序把这一组路径中多余的片段对应的点删掉，如此迭代到不动点。

为了满足和原本的地图完全一致，我们需要保证这些条件：

- 字典的遍历顺序和原始键顺序相同，因为 PrunePaths 依赖这个顺序。
- 同一个键下的片段需要保持插入顺序，因为 UnstableShuffle 的就地打乱会依赖初始顺序。同时，插入时会做重叠检测，所以插入顺序也决定了哪些路径片段会被丢掉。
- 每一组的访问次数和每组的大小都不能改变，因为每一轮都会消耗随机数。

3 第一次优化尝试

我首先试着在原始算法的基础上尽可能压缩常数。

3.1 键的打包

字符串拼接会带来巨大的动态内存分配开销。所以，我们把字典的键改为定长的 POD：每个值放进一个 3 bit 的槽位（因为只有八个地图类型），而最多 17 个点，因此整个键可以打包进一个 uint64_t。

为了使得整数上的比较和原始键值上字典序的比较给出相同的结果，我们会查表映射一次，按照 $0 < 1 < 10 < \dots < 17 < 2 < \dots < 9$ 的顺序编入。先古之民的特例永远排在 0 和 1 之间，因此为其插入一个特殊的值即可。这些编号是从 1 开始的，因为空白的位置是 0。排序只在一轮结束时对几百个键做一次，所以这里用一个稍慢的比较换一个更快的发射路径是划算的。

字段布局（高位在前）：

位	含义
[63:61]	起点类别：列 0 为 0，先古之民为 1，其余为列 + 1
[60:54]	（起点行, 长度）的索引，二元组可以压进 7 bit
[53:51]	终点列
[50:0]	每个点一个 3 bit 槽位，类型 - 1，高位在前

3.2 用 DFS 代替枚举所有路径

一条子路径可能属于上十条不同的路径，会被插入字典很多次。原始实现枚举了全部路径，但这完全是不必要的，它只需要保证每条不同的片段插入一次就可以了。因此我们改为以每个分支点为起点的 DFS：进入一个点时，就把它的类型压入键中，遇到合法的终点（父节点数不小于 2）就 emit 一次，回溯时弹出。这样每个不同的片段只被 emit 一次，而键内部的顺序也与原始实现完全一致。

剪枝可能留下一些不再通向 boss 的孤立点，而 DFS 遍历不再考虑它们。

3.3 片段的内存压缩

原始实现给每个片段复制一份 `MapPoint*` 数组，并用指针比较来做重叠检测。一次 pass 会插入约一千个片段，平均长度约为 11，也就是每轮会有 90 KB 左右的内存访问。但是，这些数据里只有极少数最终会被 `PrunePaths` 读到：真正的重复片段每轮只有三组左右。

为了比较重复路径，我们并不需要完整的行和列。片段必然在每一行上恰好有一个点，而既然键已经确定了起点、终点和长度，它内部的点就完全由每一行的列号决定。所以，和之前类似，整个片段可以编码进一个 `uint64_t`：每行一个 3 bit 的列号，因为地图只有 7 列。

这样下俩，重叠检测就变成两个整数按照 3 bit 字段逐段比较，无需方寸。字典的值也从一个 vector 变为了 8 字节，底层的 48 bit 存放首片段的列编码，而高 16 bit 存放重复列表的下标。由于重复极其稀疏，远远少于 2^{16} ，这样做是足够的。

3.4 展平的遍历视图

每有一条路径经过一个点，DFS 就会进入它一次，每一次剪枝总计约 3000 次。原始的 `MapPoint` 里，父节点和子节点各是一个独立的堆上 vector，每次进入要街引用 2 到 3 个指针。

因此我们在 `route_scan` 之后压缩整个地图：每个点只需要 6 字节来记录子节点区间、列号、类型索引，外加一个标志位。我们还需要一份存储与 boss 房连通的子节点列表。它们加起来约 1.3 KB，可以常驻 L1。同时遍历的运行状态改为值传递，以减少访存。

3.5 专用的哈希表

上述内容做完后，`build_segments` 里约 90% 的时间都花在哈希表的查找/插入上：一次 pass 约 1700 次探测（其中约 1000 次是新键），而只有约三组键真的有重复。所以我们将 `ankerl::unordered_dense` 换成了一张为这个场景手写的特殊哈希表 `SegmentTable`，每个线程有一个单例，由所有种子共用。

它将键与值放在同一个 16 字节的槽里，一次探测只需要访问一条缓存行，而 `unordered_dense` 需要两次。空槽则用 `key=0` 表示，因为真实的键永不为零。

3.6 效果

效果很明显，比原始的大概 13ms 一次加速了几十倍。然而，最后的速度是 18.3 μ s 一次，其中超过 90% 是哈希表的依赖访存。这时就已经没有提升空间了。

另一个明显的想法是直接插入 vector 并排序。这是 $O(n \log n)$ 的，反而更慢。

4 第二次优化尝试

我们在前面已经说过，重复的键只有三个左右。为三个重复遍历一千多个键太浪费了，是否有算法可以直接找出它们呢？

实际上，如果仔细通读一遍，会发现原始算法是在找 DAG 上的菱形结构，然后删掉它。而既然是菱形，很容易想到通过分叉点和交汇点定义。我们下面将它形式化。

4.1 菱形搜索

地图是一个 DAG，它的点按层排列：第 0 行是先古之民，第 R 行是 boss，每条边必须从第 r 行指向第 $r+1$ 行。我们只考虑能够从先古之民出发到达 boss 的路径上的点。

如果 u 的出度 ≥ 2 ，就将其称作分叉点；入度 ≥ 2 ，就称作交汇点。原始的键可以表示为 (u, v, L, τ) ，其中 u 是分叉点， v 是交汇点， L 是长度，而 τ 是路径上所有房间的类型。这里的 L 是多余的，我们已经在第一次优化中提到过了。

如果两个键相同的片段在非首尾的行上选择了同一个列，那么它们重叠。接下来，我们要提出判断重叠的算法。

显然，不同键之间互不影响，所以我们可以找到所有的分叉点 u 依次处理。我们可以考虑从 u 开始进行 DP，来收集不重复的路径。

我们不妨将一对不重复的路径看做二元集合 $\{a, b\}$ ，其中 a, b 是同一行上的终点。方便起见，令 $C(p)$ 为 p 的子节点集合， $t(p)$ 是 p 的房间类型，而 r_p 是 p 所在的行。

设 S_r 是第 r 行上的 DP 状态，也就是路径集合，那么初值显然是

$$S_{1+r_u} = \{(a, b) \in C(u)^2 \mid a \neq b, t(a) = t(b)\}$$

而状态转移方程是：

$$S_{1+v} = \{(a', b') \mid \exists (a, b) \in S_v, a' \in C(a), b' \in C(b), a' \neq b', t(a') = t(b')\}$$

终点 v 必定是交汇点：

$$\exists (a, b) \in S(r_v - 1). v \in C(a) \cap C(b)$$

不过，这只是路径终点的必要条件，这是因为原始算法是依赖搜索顺序的。我们最后需要验证一下它们真的是不重叠的。而且，还需要将它们按照原始算法顺序排好序，不过这个顺序就是 DFS 序。

5 性能测试

这是单线程下 20000 个种子的数据。

阶段	原始算法	菱形搜索	加速
seed_profile	10.54 s	3.10 s	3.4x
map_prune	8.46 s	1.13 s	7.5x
build_segments	7.91 s	0.90 s	8.8x
build_segments:route_scan	0.29 s	0.13 s	2.2x
prune_paths	0.066 s	0.059 s	—
map_layout	1.26 s	1.24 s	—
map_generate	0.34 s	0.33 s	—
map_assign	0.26 s	0.25 s	—

现在一个种子的地图生成速度来到了大概 0.15 ms。

5.1 剩余瓶颈

剪枝不再是大头之后，剩下的是 map_layout（约 40%）。它按行反复迭代，每个点每次迭代都要建一个 `std::set<int>` 来表示“允许移动到的列”，这是明显的浪费：7 列只需要一个位掩码。这是下一个值得动的地方。

build_segments 内部的余量则不大：一次 pass 只有十几次分叉点、三四个候选终点，剩下的是几百个很小的内存访问。真正还有量级空间的是**重建次数**（每成功剪枝一次就要整轮重算，一张地图平均 7.2 次）。这要求剪枝状态可以增量维护（片段集合、插入顺序、以及每次 pass 的随机数消耗都要一致），而现在这种做法下反而更难：每次重建的成本已经降到原来的八分之一，增量化的收益也随之变小。