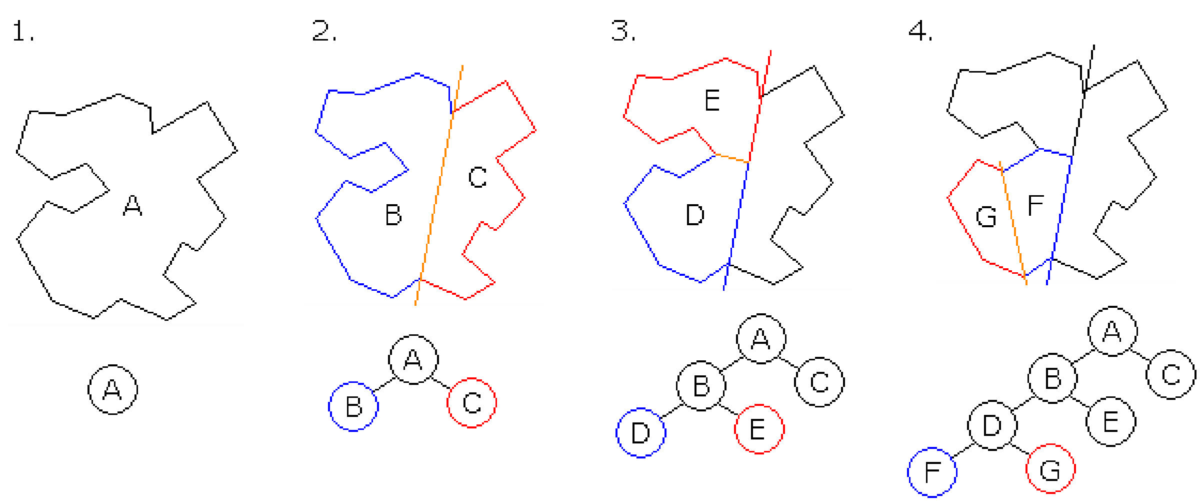
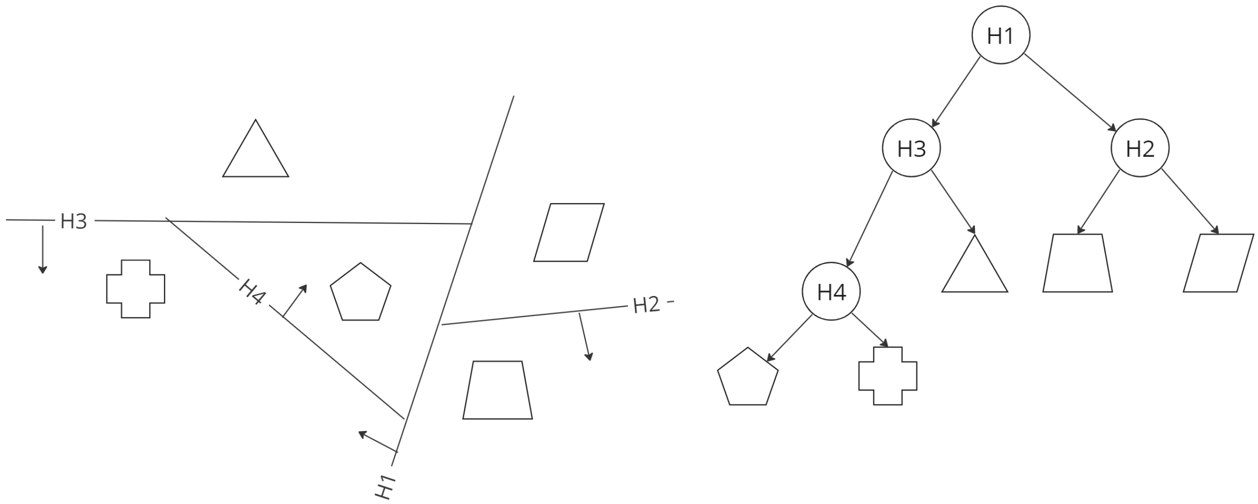
BSP Trees

BSP Trees are a generalization of kd trees. The main difference is that in KDtrees we splitted the axes in parallel with the axis now we want to relax this condition and allow splitting in any direction



the main idea is to introduce a splitting plane and a direction, then assign to the children either a node containing a geometry if there is only one in the subspace generated or another BSPTree. the left child always indicates the inner side of the splitting, the right one the outside



more properly we can give a **definition:**

given a set of polygons $S = \{p \in \mathbb{R}^d\}$, and a plane $H \in \mathbb{R}^{d-1}$, we can define H^+ as the positive halfspace and H^- as the negative halfspace divided by H

if $|S| = 1$, S is a leaf node v, storing $S(v) = S$

if $|S| > 1$, S is an inner node that stores:

- H_v, the splitting plane
- $S(v) = \{p \in S | p \subseteq H_v\}$ the set of polygons completely included in H_v
- T^+, the subBSP over $S^+ = \{p \cap H_v^+ | p \in S\}$, i.e. all the polygons in the positive halfspace
- T^-, the subBSP over $S^- = \{p \cap H_v^- | p \in S\}$, i.e. all the polygons in the negative halfspace

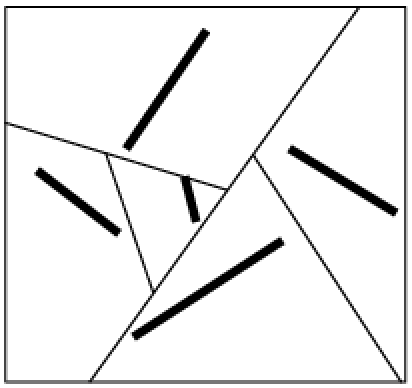
NOTE: the $p \cap H_v^{\pm}$, w.r.t. polygons included in an halfspace are called **Fragments**

we can also define in a similar fashion to the KD trees the **region of a node** v, $R(v)$, being the convex subset of R^d covered by v, i.e. the region of space where the node exists. notice that the region of a node is always a subset of the region of its parent

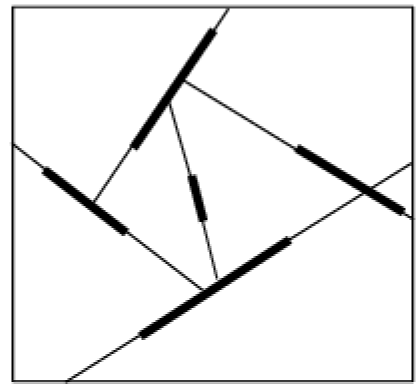
and we can also define a **Supporting plane**, as a plane that completely contains a polygon

Autopartition

a BSP where all the splitting planes H_v are supporting planes)



general (unrestricted) BSP



auto-partition

NOTE: the order in which we split the plane matters, since it could lead us to different trees

Construction(in 2d)

```
def autopartition(S=set of lines in R^d):
  if |S|<=1:
    T= leaf v containing S
  else
    chose s1 in S in as spliting line
    let LS1=supporting line of S1
    compute s+ = {s in LS1+}
    compute s- = {s in LS1-}
    calculate T+=autopartition(S+)
    calculate T-=autopartition(S-)
    T=node v containing T+ and T-, LS1 and S(v)
  return T
```

The algorithm can be randomized by randomizing S in the beginning

Complexity

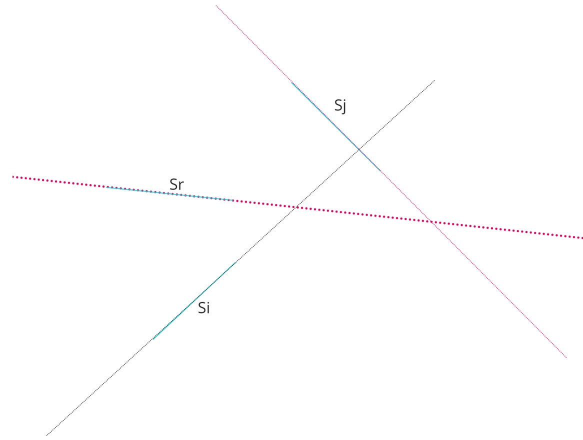
in 2d, $n = |S|$, we expect a number of fragments in $O(n \log n)$ constructed in a time $O(n^2 \log n)$. to show this we can start by saying that the number of fragments, being the size of the BSP is equal to $\bar{n} = \sum_{\nu} |S(\nu)|$ and since for every node we split in half the number of leaves$\#nodes = 2\#leaves - 1 = 2\#fragments - 1 = 2\bar{n} - 1$

now let s_j be a segment not yet **"consumed"**(used) by the algorithm and s_i the **currently chosen segment.**

Now we ask: **what is the condition for s_j to be split by the supporting plane of s_i?**

take this case:

if we selected s_r before s_i, s_j whould have been **Shielded** by the supporting plane of s_r



Driven by this we can define a new **Distance**:

$$\text{dist}(s_i, s_j) = \begin{cases} \#\text{segments } s_r \text{ with interc. point } L_{s_r} \cap L_{s_i} & if \quad L(S_i) \text{ intercepts } S_j \\ +\infty & else \end{cases}$$

i.e. when we have an interception between the supporting plane of s_i and the segment s_j this distance means the number of "shielding" fragments between s_i and s_j

we can say that L_{s_i} splits $s_j \iff i = \min\{i, j, j_1, \ldots, j_k\}$, where k is the number of segments between s_i and s_j

since we are randomizing we have to take in play probability, and we will say that the probability $P_r[L_{s_i} \text{ splits } s_j] = \frac{\#permutations(j_1,\ldots,j_k)}{\#permutations(i,j_1,\ldots,j_k)}$, i.e. the number of "good cases" among all the cases, where the good cases are all the cases where i is the minimum, and the number of permutation is one element less, so we obtain $P_r[L_{s_i} \text{ splits } s_j] = \frac{(k+1)!}{(k+2)!} = \frac{1}{k+2} = \frac{1}{dist(si,sj)+2}$

now we can calculate the expected number of splits caused by the segment S as the sum of all those probabilities, so $\sum_{s' \neq s} \frac{1}{dist(s,s')+2} \leq 2 \sum_{i=0}^{n-l} \frac{1}{i+2} = 2\log(n)$, where the $\leq 2 \times \ldots$ is because every split occours at most 2 times, one per direction the line can follow.

at this point, since we have to compute this for every of the n segments we expect to compute at most $2n \log n$ splits, and since we started with n fragments we end up with a number of segments in the order of $O(n + 2n \log n) = O(n \log n)$

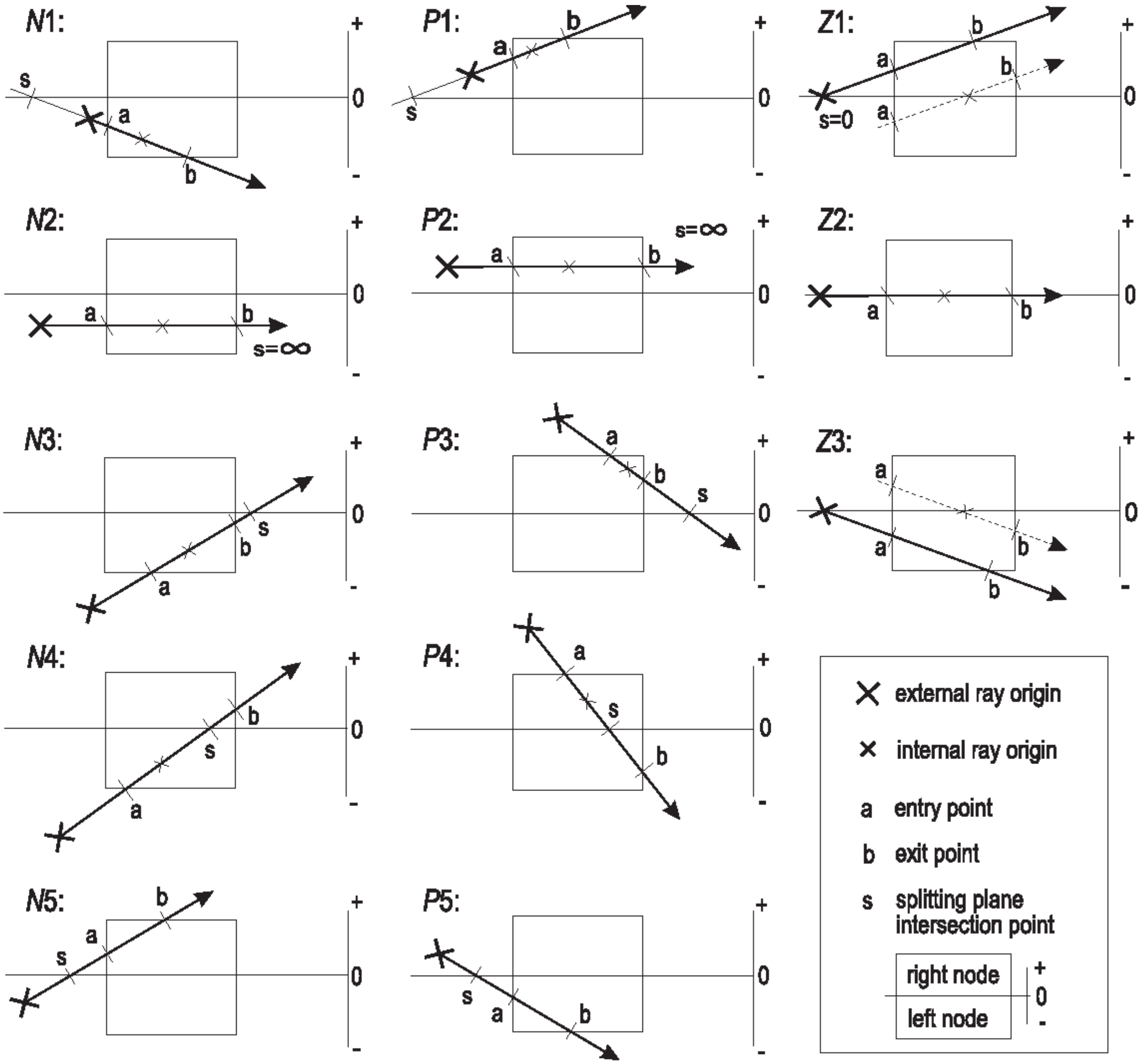
now since the number of recursive calls is equal to the number of fragments and we have to compute the possible splits for every fragments we end up with n times the time for building all fragments, since we have$O(n \log n)$ fragments, each one built in $O(1)$ time, we end up with a time complexity of $O(n^2 \log n)$

Construction(in 3d)

in this case the construction is $O(n^2)$

Example applications

Ray casting



Rendering a set of polygones without a z-buffer(painter's algo)

Quality of the BSP

Balancing Vs Splits

if for example we need to make a **classification** task we need to reduce the worst case behaviour, that means reducing the depth of the tree, meaning that we want to **Balance the splits** if we want to make **depth sortings** we are interested in reducing the number of fragments, meaning that we need to **reduce the size**

the question: **Can we measure the quality of the BSB?**(Spoiler: Yes!)

we can measure the cost of a BSP:

$C(T) = 1 + P^- C(T^-) + P^+ C(T^+)$, where:

1 is given by the fixed cost that we need to spend at runtime if our query intercepts a node

$P^{\pm} = \frac{vol(R(T^{\pm}))}{vol(R(T))}$ is the probability that the traversal at runtime needs to enter in the negative/positive subtree

$T^{\pm}$ are the positive/negative subtree

Distribution Optimized(Self Organizing) BSP

we start with a **Autopartition BSP**, and we have a ray that starts somewhere and we want to shoot it into the scene and see if it has any interceptions.

The cost is equal to the number of visited nodes, that is less than the depth of the tree d times the number of stabbed leaves

$$C(T) = \#nodesVisited \leq depth(T) \cdot \#stabbedLeaves$$

to notice that in an autopartition the leaves always have an empty space inside, this means that the stabbed leaves are the spaces in which the ray passes.

what do we want to minimize is the number of stabbed leaves until the ray hits a polygon

we need to consider a probability density function $\omega : D \to \mathbb{R}$, where $D \supseteq \mathbb{R}^5$ is the space of the rays, let $l \in D$ be a ray, defined by a starting point and a direction, and S the set of polygones of the scene, let $p \in S$, we define the $score(p) = \int\limits_D \omega(l) w(p,l) dl$

where $w(p,l)$ is a **weight** that aims to capture the probability that a specific ray $l$hits the polygon p and is influenced by the angle between ray and polygon, we then need the normal of the polygon n and the direction of the ray l_d

$$w(p,l) = |n \cdot l_d| \cdot \frac{area(p)}{area(s)}$$

with this we can improve the BSP construction: instead of choosing a random polygon we can sort S by $score(p, l)$ and take the ones with the highest score

what do we do is **augmenting the BSP**

Augmented BSP-Tree

we are going o turn the BSP into an on-deman construction, trying to move the polygons that are hit more frequently into the top of the BSP, in this way they will be hit earlier.

in this version each node v will store:

- the splitting plane H_v
- a set of polygons P_v
- potentially the region $R(v)$, but it's not really needed

we will say that if the node is a **preliminary leaf** we will store a list $L(v) \subseteq S$ of polygons associated with the node v

we will also store a **visit counter** $T(v)$ for each node that stores how many times we have visited that node

we will also store a **list counter** $T(p)$ for each polygon that stores how many times a polygon has bit hit

```
def testRay(ray l, node v):
  if v is leaf:
    increment t(v)
    test l against all L(v)
    increment T(p) for all polygons in L(v)that are hit
    if T(n)>threshold:
      subdivide v
    return the hit point or none
  else:
    v1 = child of v on the same side of the starting point of l
    hitPoint = testRay(l, v1)
    if no hit in v1:
      hitPoint = testRay(l1, v2)
    return hitPoint
```

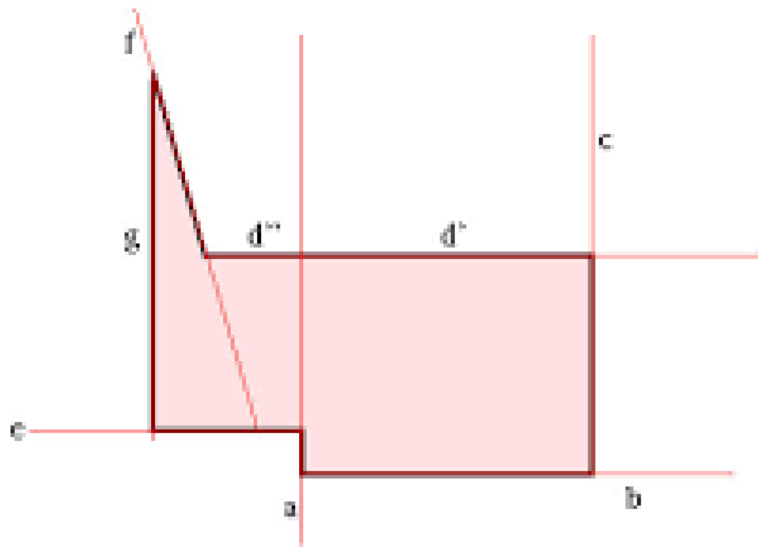
the subdivision step of preliminary leaves

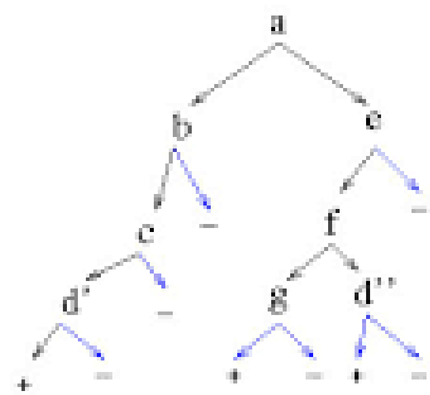
When: split if $T(v) > treshhold$

How: $T(p)$ is the list of hits on the polygons(**hit counter**) and we increment it when we find an hit. if we want to split, we take the polygon p^* with the maximum counter and maintain L as a heap

Object representation using BSPs

in this scenario the leaves has a different meaning, the one of ***"inside"*** and ***"outside":***

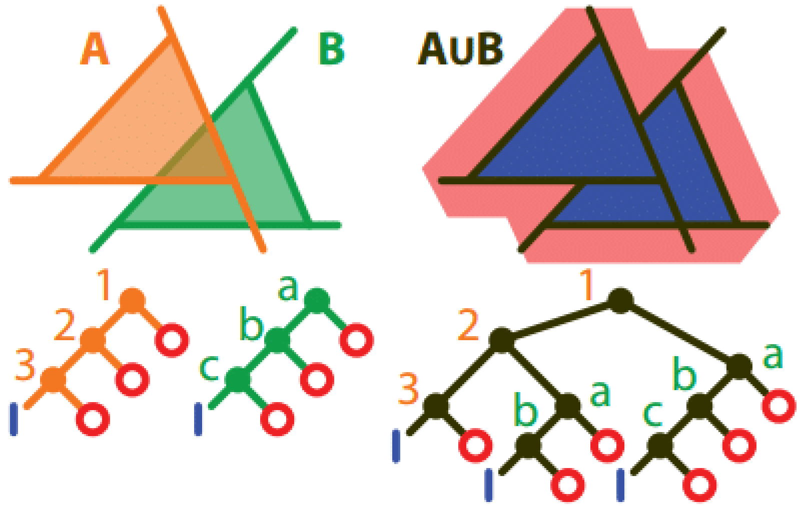




Merging BSPs

given 2 BSPs BSP_1, BSP_2 and an operation $\circ \in \{\cap, \cup, \setminus\}$, we want to compute

$BSP_1 \circ BSP_2 \rightarrow BSP_3 : \forall \text{ leaves } l_3 \in BSP_3 l_3 = \{c_1 \circ c_2, |c_1 \in l_1, c_2 \in l_2\}$



ANSATZ

we start with a BSP T and a plane H containing the polygon $p_H \subseteq H$ that lies completely in it, we search for a new BSP $\hat{T}$ with H in the root

we can create a function $\text{partition-tree}(T, H) \rightarrow \hat{T}$, that calculates from a subroutine the subtrees $(\tilde{T}^+, \tilde{T}^-) = \text{splitTree}(T, H, H)$, then $\hat{T} = (H, p_H, \tilde{T}^+, \tilde{T}^-)$

$\text{splitTree}(T, H, P) \rightarrow (\tilde{T}^+, \tilde{T}^-)$

$\tilde{T}^+ = T \cap H^+$, $\tilde{T}^- = T \cap H^-$

T is the root of a BSP $= (H_T, p_T, T^-, T^+)$

H is a splitting plane

$P = H \cap R(T)$

if T is leaf: $(\tilde{T}^+, \tilde{T}^-) = (T, T)$

otherwise T is an inner node

If H and H_T are coplanar with opposite normals: $(\tilde{T}^+, \tilde{T}^-) = (T^-, T^+)$

If H and H_T are not coplanar, but facing the same direction: $(\tilde{T}^+, \tilde{T}^-) = ((H_T, p_T, T^-, splitTree(T^-, H, P)^+), (H_T, p_T, T^+, splitTree(T^+, H, P)^+))$

and analogue for all possible orientations

mixed case:

cool material

https://slideplayer.com/slide/13535755/

https://www.semanticscholar.org/paper/Finding-perfect-auto-partitions-is-NP-hard-Berg-Khosravi/666c26350fbc2a6130d4c7a78afae5f8fdae07ae

https://www.semanticscholar.org/paper/Coupled-Use-of-BSP-and-BVH-Trees-in-Order-to-Ray-Cadet-Lécussan/3db1fb59a1409a4cacad5a9060710b68fb9b1ce7

https://www.researchgate.net/figure/The-construction-of-a-simple-BSP-in-2D-On-each-step-the-space-is-divided-in-two_fig13_287646188

https://www.semanticscholar.org/paper/Fast-Robust-BSP-Tree-Traversal-Algorithm-for-Ray-Havran-Kopal/877db116e624eb917519ae430a6ff08f1604cf48

https://onlinelibrary.wiley.com/doi/10.1111/1467-8659.t01-1-00586

https://commons.apache.org/proper/commons-geometry/tutorials/bsp-tree.html

https://www.researchgate.net/figure/4-Plane-based-Booleans-are-performed-by-merging-BSP-trees-8_fig12_346013718