I would like to summarize what I have implemented for spill trees and know your opinion. So, we can decide how to continue.

Actual implementation of Spill Trees, is based on the original implementation of Binary Space Trees.
I start with the list of all points and, in each step, I split it according to the midpoint split.
All points to the left of the midpoint are included in the left child, and all points to the right of the midpoint are included in the right child. I expand the bounds of each child with their points.
Then, I consider the points in the range \((\text{midpoint} - \tau, \text{midpoint} + \tau)\).
If it is possible to build a overlapping node (according to the constraints of hybrid spill trees), I include the points in the range \((\text{midpoint} - \tau, \text{midpoint})\) in the right child, and points in the range \((\text{midpoint}, \text{midpoint} + \tau)\) in the left child. But, I don’t expand the bound of the children with these points, so we can properly calculate the score of each children when traversing the tree.
If I expanded the bound with them I couldn’t calculate the appropriate score. See an example in Figure [1].

Figure 1: In black color: bounds of children nodes. In cyan color: the real bound including shared points. \(p_q\) represents the query point. If we consider the real bound, we couldn’t decide which node to traverse. If we don’t consider the shared points, it becomes clear that we have to traverse the node on the left.
In this way, I can continue with the common approach to calculate the score when traversing the tree. In mlpack, KDTrees are built using the midpoint split but, when calculating the score and deciding how to traverse the tree, instead of the distance to the hyperplane determined by the midpoint split, the distance to the hrect bound is considered because it provides a tighter bound. Also, I suppose this approach is required in order to do dual tree search. So, I decided to take the same approach for spill trees: build the tree splitting the points with a midpoint split, and considering the bounds when calculating the score and traversing the tree. But this implementation is different to the approach mentioned in Ting Liu’s paper, where it is supposed that you consider the same hyperplane when splitting the list of points and when deciding which node to visit in a dfs search. So, the difference is that, we consider a midpoint split hyperplane when building the tree, but we consider a different decision boundary when traversing the tree (the decision boundary is defined by the set of points with the same distance to the left child’s bound and the right child’s bound). If we analyse this difference, we will realize that it can make a difference in the overlapping of nodes.

Figure 2: Examples of different situations, to show the difference between the midpoint split and the decision boundary.
With normal spill trees, we can guarantee that all nodes at a distance less than $\tau$ of the query point $p_q$ will be considered by the defeatist search. However, with our implementation, we can guarantee that all nodes at a distance less than $\tau/2$ of the query point $p_q$ will be considered by the defeatist search.

So, it looks like we have an overlapping of $\tau/2$ instead of $\tau$, as it can be seen in Figure [2].

Let’s prove that: “all the points at a distance less than $\tau/2$ from $p_q$ will be analysed”:

Let $p_q$ be a point to the left of the decision boundary (the opposite case is similar). This means that we will choose the left node when doing defeatist search.

Let $N_r$ represent the right child and $N_l$ the left child.

We define: $S_r = \mathcal{D}^p(N_r) - \mathcal{D}^p(N_l)$

$S_r$ represents all the points in the right node that are not in the left node.

So we want to prove: $\forall p_r \in S_r : dist(p_q, p_r) > \tau/2$

Figure 3: Different possibilities for a point in the left child node.

We can partition the proof in 3 possible cases (Figure [3]):

- (A) $p_q$ is to the left of the midpoint split:
  
  Clearly: $\forall p_r \in S_r : dist(p_q, p_r) > \tau/2$

- $p_q$ is to the right of the midpoint split, and to the left of the “decision boundary”:
  
  We know that all the points in the left node’s bound, are to the left of the midpoint split, so:
  
  - $dist(p_q, leftbound)$ represents the minimum distance from $p_q$ to the left bound.
  - $dist(p_q, rightbound)$ represents the minimum distance from $p_q$ to the right bound.
  - $dist(p_q, midpoint)$ represents the minimum distance from $p_q$ to the midpoint hyperplane.

  So we can assert that:
  
  $$dist(p_q, leftbound) \geq dist(p_q, midpoint)$$

  Also, as $p_q$ is to the left of the “decision boundary”:
  
  $$dist(p_q, rightbound) > dist(p_q, leftbound)$$

  So, by transitivity:
\[ \text{dist}(p_q, \text{rightbound}) > \text{dist}(p_q, \text{midpoint}) \]

Results in: \( \forall p_r \in S_r : \text{dist}(p_q, p_r) > \text{dist}(p_q, \text{midpoint}) \) (1)

Let’s consider 2 cases:

- (B) \( \text{dist}(p_q, \text{midpoint}) \leq \tau \):  
  All points in \( S_r \) are to the right of the hyperplane \( \text{midpoint} + \tau \), so:  
  \[ \forall p_r \in S_r : \text{dist}(p_q, p_r) > (\tau - \text{dist}(p_q, \text{midpoint})) \]
  So, with (1):
  \[ \forall p_r \in S_r : \text{dist}(p_q, p_r) > \max((\tau - \text{dist}(p_q, \text{midpoint})), \text{dist}(p_q, \text{midpoint})) > \tau/2 \]
  ((B) is the interesting case that imposes \( \tau/2 \))

- (C) \( \text{dist}(p_q, \text{midpoint}) > \tau \):  
  Because of (1):
  \[ \forall p_r \in S_r : \text{dist}(p_q, p_r) > \text{dist}(p_q, \text{midpoint}) > \tau > \tau/2 \]

So, we should define which approach we want to adopt:

- Continue with actual approach, which is different to the mentioned in the paper:
  - Has some advantages:
    - The implementation is very similar to the implementation of BinarySpace-Trees in mlpack.
    - We can reuse neighbor search rules and most of the code of knn search.
    - Using bounds instead of cutting hyperplanes can provide tighter bounds. This can make a difference when calculating the score and doing backtracking on non-verlapping nodes. In fact, I suppose this is the reason because actual kdtree implementation uses bounds.
  - And disadvantages:
    - We can see some differences between our implementation and the implementation proposed in the paper, which can be a disadvantage if we want to benchmark and make assertions about the performance of spill trees in general.
    - With a given value of \( \tau \), we can be sure we have a overlapping of \( \tau/2 \) according to the decision boundary, not \( \tau \).

- Implement a similar approach to the one mentioned in the paper. Consider midpoint cutting hyperplanes when calculating the score, the same hyperplanes that were considered when splitting the node. This mean the “decision boundary” and the midpoint split hyperplane will be the same.
  - Has some advantages:
* It will have an implementation similar to one proposed in the paper, which is an advantage if we want to benchmark and make assertions about the performance of spill trees in general.

* With a given value of $\tau$, we can be sure we have a overlapping of $\tau$ according to the decision boundary.

- And disadvantages:
  
  * We would have to reimplement spill trees, we cannot reuse BinarySpace-Trees.
  
  * We can’t reuse neighbor search rules. So, we would have to create a different implementation for spill trees.
  
  * Possibly, this could result in less tighter bounds and less number of prunes.
  
  * We should analyse if this could be used for dual tree search, I am not sure this is possible.

In case we decide to implement this option, I can think of 2 possible approaches for the implementation:

  * We could modify actual implementation to record the midpoint value and the dimension and then use that information to compare the query point and decide which child node to traverse first.

  * We could consider cutting hyperplanes that are not necessarily parallel to an axis. For example, as suggested in the paper, we could approximate the pair of furthest points $p_1, p_2$, in the current set of points. Then, split them according to the projection over the vector defined by $p_1, p_2$. We could record that vector in each node, so we can project the query node and decide which child node to traverse first when doing dfs. This is the approach implemented in: [ref]