Reasoning Behind TreeWorks Implementation

1 Why predicates not functions?

First, we should be aware of the fact that predicates in C++ are just ordinary functions that return logical value (C++ type `bool`). Hence, from the programmer perspective there is hardly any difference, while from the library implementer standpoint this mechanism offers several advantages:

- end-user must focus only on the logic of `select` function and not on the low-level details, specific to implementation,
- finer control over correctness of the code, for instance, implementing `select` function through predicates guarantees that only children nodes will be considered (i.e. problem of accidental or intentional inclusion of non-children nodes in the search path is avoided),
- formalizing `select` function with well defined mathematical entity provides opportunities to the library developers to implement high and low level optimizations in the `treeSearch` function.

Consider `select` function given in the paper as example III.B. The original code is given below:

```
select($u, \mathcal{K}$) :
 1: if $u$ is a leaf then
 2:     return ($u$)
 3: end if
 4: $returnList \leftarrow$ NULL
 5: $childList \leftarrow$ children($u$)
 6: for all $v \in childList$ do
 7:     if $k_v$ intersects with $\mathcal{K}$ then
 8:         add $v$ to $returnList$
 9:     end if
10: end for
11: return $returnList$
```

We can identify two main element here. First we decide if node u itself should be returned as a part of the result (lines 1–3). Then for each child of u we check if they should be included in the search path by checking user specific criterion (given by presence or absence of search item $\mathcal{K}$). The above is naturally described by the following questions:

1. Should I include u in the result?
2. Does given node contain search item $\mathcal{K}$ (i.e. should it be included in the search path)?

Suppose that the first question is represented by predicate $p(u)$, which is true iff u should be in the result. Then, let describe the second question with predicate $q(u)$, which is true iff u contains $\mathcal{K}$. Obviously, to decide which nodes will be included with respect to predicate q we have to iterate over all children nodes of node u. This task will be repeated always, and it is hard to imagine situation in which user can be interested in writing this loop all-over-again. Consequently, it is desired to hide this loop from the user requiring only predicate q to be implemented. Interesting question is: should we distinguish between predicate p and q, i.e. is it possible that node that does not satisfy p will be included in the search path?

Using the above argumentation and corresponding predicate formulation function, `select` can be replaced with simpler function:

```
query($u$):
  if $u$ intersects with $\mathcal{K}$ then
    return TRUE
  end if
  return FALSE
```

Clearly, `query` function focuses on the logic of the problem, leaving details like, e.g. iterating other children nodes, to the library implementers.