RECURSIVE STRATEGIES

Given a permutation class $\mathcal{C}$ a *proof strategy* for a tiling T returns a disjoint set of tilings $T_1, T_2, \ldots, T_k$ such that the union of these individually intersected with $\mathcal{C}$ is the tiling T, i.e.

$$T \cap \mathcal{C} = (T_1 \cap \mathcal{C}) \sqcup (T_2 \cap \mathcal{C}) \sqcup \ldots \sqcup (T_k \cap \mathcal{C}).$$

If the proof strategy returns a single tiling we call is an *equivalence* proof strategy. If returns more than one we call it a *batch* proof strategy.

For a permutation class $\mathcal{C}$, consider the tree where the root node is the 1×1 tiling with $\mathcal{C}$. The children of a node are the tilings obtained a proof strategy on it, thus form a disjoint union for the node while working inside $\mathcal{C}$. If all the leaves of the tree are subsets of $\mathcal{C}$ then they define a cover for $\mathcal{C}$, while the tree itself is a proof certificate of this cover. We call such trees *proof trees.*

Here is a rough outline of how we propose to search for proof trees.

Remark 1 (Outline of basic proof tree search)*.* In order to find a cover for $\mathcal{C}$ we create an and-or tree. The root node is the 1×1 tiling with $\mathcal{C}$ and originally set to False, as we reject the trivial cover. For a node with tiling T, and for each proof strategy we add to his children the tilings from the proof strategy and conjoin the edges to these. When a tiling is a subset of $\mathcal{C}$, it is marked as True and this information is propagated through the tree. When edges are conjoined, this represents an “and”, therefore in order for a node to be True then a set of its children which are conjoined must all be True. The “or” part of the tree is that we need only one set of conjoined children to be True. When the root node becomes True then we know that the search tree contains a proof tree for $\mathcal{C}$.

We will do this in the following way, we will create OrNodes, which will consist of single tilings. From a tiling/OrNode, first we create the SiblingNode, which will consist of all tilings reachable by equivalent proof strategies, we call these *sibling* tilings. In order for the SiblingNode to be verified we require that one of the sibling tilings to be verified.

For each batch strategy we create an AndNode which will consist of the set of OrNodes with tilings corresponding to the strategy. In order for an AndNode to be verified, all of the OrNodes in its set must be verified.

The remainder of this document will discuss recursive strategies, in an informal way. (For want of a better name) A *recursive* strategy (these don't fit the definition of strategies given above) for a tiling, is a set of subtilings (currently disjoint), that if the enumeration/permutations are known for, then the enumeration/permutations are known for the tiling. There are three ways for a subtiling to be verified

- it is verified as subset of $\mathcal{C}$,
- it points to a verified node elsewhere in the tree,
- or it points to an ancestor of the tiling.

Here is how we propose to find such proof trees relying on recursive strategies while maintaining the original proof trees definition. We will still use the OrNodes, AndNodes and SiblingNodes described above, but now keep track of more information.

Class 1 (OrNode)**.**

- a tiling T,
- a SiblingNode - we will discuss this more when introducing SiblingNodes, but essentially it stores reachable siblings of T,
- the set of ancestral OrNodes,
- parent AndNodes - (the strategies it belong too),
- children AndNodes - (the batch strategies from an T).

We realized while trying to propagate ancestral verification the lack of knowledge of ancestors made things complicated. Therefore as you can see we have adopted the approach of storing ancestral OrNodes as we go.

Note since we are storing the ancestral OrNodes, a tiling T will belong to multiple OrNodes. However, the work done a tiling T will only ever be done once and stored in the cache.

Class 2 (AndNode)**.**

- a parent OrNode with tiling T,
- a set of OrNodes corresponding to a batch strategy on T,
- formal step explaining the corresponding batch strategy
- a set of ancestral OrNodes.

Class 3 (SiblingNode)**.**

- a set of OrNodes with equivalent tilings,
- a map explaining the equivalences,
- a set of ancestral OrNodes.

Let's discuss what we do when we expand the search tree. We first find an unverified OrNode x with tiling T to be expanded. For each batch strategy from the tiling T, we create an OrNode with each of the strategy's tilings and then an AndNode with them all.

When we create the OrNode for a tiling T, we also create a SiblingNode with each of the equivalent proof strategies on the tiling T.

Now, consider a recursive strategies S for T in OrNode x. For each T' in S if the tiling is not in the cache we create the OrNode but do not expand it until it is found naturally. If it is in the cache and one of its OrNodes is verified, it is verified. If it is verified by the verification check, it is verified. The final case is if it points somewhere unverified, say node y. We make a new OrNode with T' if and only if y is in the ancestral set of node x, and mark it verified. All of the tiling's OrNodes in S are then connected together in an AndNode. We can propagate verified in the normal way.

Note, we believe this removes the need for different types of verified. The challenge then comes from keeping track of ancestral nodes, and for combining siblings in SiblingNodes.