Introduction to *In-Silico* learning
Data Mining Algorithms

Abstract

This tutorial will show how disparate Data Mining Algorithms are just specific instantiation of general data mining approaches. After sketching our use case scenario as well as a generic setting for data mining algorithms, we're going to analyse how to generate association rules from a given dataset. We're going to decompose this problem in two parts (return the most frequent supported patterns and then inferring association rules) using specific heuristics for speeding up the computation.

1 Use Case

Now let us look at a completely different use case: now, we want to determine which strategy (*gameplay hypothesis*) players are adopting to win battles. However, given the very compressed nature of the data provided in the previous tutorial, it is not possible to extrapolate any possible game patterns and correlate it to a battle victory. This is because the above data does not contain specific information to a single battle, as it only contains only actions that consider the whole gameplay, nor does it provide a possible outcome of the campaigns. At this point, we are forced to change datasets, and then to change video games.

Clash Royale is a Real Time Strategy game combining collectible card games, tower defense, and multiplayer online battle arena. Battles' strategy are driven by cards: players must construct a deck composed of eight cards, which are used both for attack and for defense actions. Such cards can be either bought in shops, can be won during a battle, or can be requested to a clan member. When a battle starts, the game randomly picks a set of four cards, which will be the only ones the gamer is going to use during the battle. In this scenario, we want to implement a **recommendation system**, suggesting the player which is the best (set of) card(s) to be achieved to win the next battle given the current deck configuration. This recommendation system will help novices to build up their personal strategy: similar strategies are also adopted in Turn Based Strategy games such as Civilization (e.g., which technology should the civilization learn first in order to achieve either a Cultural or Conquest Victory).

From the game APIs, we are able to dump a lot of game information, including the players' names during the 1-vs-1 battles, the 8 card deck possessed by a player before starting the battle, and who won such battle. The given dataset is available at `https://github.com/kekepins/clash-royal-analytics/`: given that there is no exact information regarding of which the four randomly picked cards were during a battle, we can approximate the current problem by considering which are the set of 8 cards that are always winning, and by trying to establish correlation rules between all the different possible winning card configurations. Albeit the task might seem easy to achieve, we can observe that extracting all the possible recommendation for 97 distinct cards turns out to evaluate $1.91 \cdot 10^{46}$ possible rules [4]. As we might see, this is a non-trivial and exponential problem to solve and we are not satisfied by returning only one single possible recommendation for initial card configuration.

In order to solve the first problem, we might consider to boil down the set of all the possible rules to only the frequent ones and, as a consequence, we need to make the running time of the algorithm dependant on the actual output size and not on the set of all the possible configurations to be returned. In fact, it might be possible that 80% of the rules are discarded after applying a minimum support of 20% and a minimum confidence value of 80%, thus making all the time involved in producing results that are going to be discarded a completely waste of time [4].

Algorithm 1 Brute Force hypothesis generation

```
function Enumerate($\mathcal{Q}, D, \mathcal{L}_h$)
    for each $h \in \mathcal{L}_h$ do
        if $\mathcal{Q}(h, D)$ then yield $h$
        end if
    end for
end function
```

Algorithm 2 Branch-and-Bound hypotheses search

```
function ($\mathcal{Q}, D, \mathcal{L}_h$)
    $Queue$:=$\{\top\}$      $Th$:=$\emptyset$
    while $Queue \neq \emptyset$ do
        $h$:=pop($Queue$)
        if $\mathcal{Q}(h, D)$ then
            $Th$:=$Th \cup \{h\}$
        else
            $Queue$:=$Queue \cup \{\, d \in \mathcal{L}_h \mid h \preceq d \,\}$
        end if
        prune($Queue$)
    end while
    return $Th$
end function
```

For solving the second problem, we do not need do adopt Machine Learning strategies that are going to return only one single configuration, but we shall adopt Data Mining ones allowing us to return multiple possible strategies, and hence returning multiple possible strategies. Furthermore, Data Mining approaches can be incrementally run over different historic data, and so we might as well achieve *continuous learning* deterministic for free.

2 Association Analysis

A specific data driven approach to generate recommendations is *association analysis*: such technique is also adopted in **market basket analysis** [4], where we want to check which are the products that a customer is more likely to buy given the elements that has already bought, in **earth science**, where we want to associate different co-occurring event patterns happening between the biosphere, litosphere, hydrosphere and atmosphere, and also in **healthcare**, where correlations can be used to analyse comorbidity condition for the elderly. While in market basket analysis one single event is a single customer *transaction*, in earth science an event could correspond to a specific time interval, and in comorbidity an event can be characterized as the cooccurrence of single patients' events within one year [1]. Yet again, in our game scenario, each event corresponds to a player's 8 deck configuration prior to each battle.

Let us simplify the problem for the moment, and let us consider that each event or transaction $t \in T$ is a given word represented as a set of unique letter occurrences (Figure 1a), and that each letter within its set is the item (Figure 1b) for which we want to provide a recommendation. Given that for the previous observations it is quite impossible to generate all the possible rules, we might as well try to decompose the association analysis task in two distinct and consequential parts:

1. **Frequent Itemsets Generation**: given a frequency/support threshold θ, we want to generate all the possible subsets of the itemset (*frequent itemset*) having a frequency/suppport greater or equal than θ (Figure 1d).
2. **Rule Generation**: for each frequent itemset Z, generate high confidence rules $X \to Y$, where each rule is composed by a binary partitioning of a frequent itemset ($X \cap Y = \emptyset \wedge X \cup Y = Z$).

Independently from the representation of $\mathcal{L}_h$, we can return all the hypotheses by enumerating all the possible hypothesis and returning only the ones satisfying a quality criterion $\mathcal{Q}$ (Algorithm 1, [3]).

TID	Itemsets
1	$\{M, O, N, K, E, Y\}$
2	$\{D, O, N, K, E, Y\}$
3	$\{M, A, K, E\}$
4	$\{M, U, C, K, Y\}$
5	$\{C, O, O, K, I, E\}$

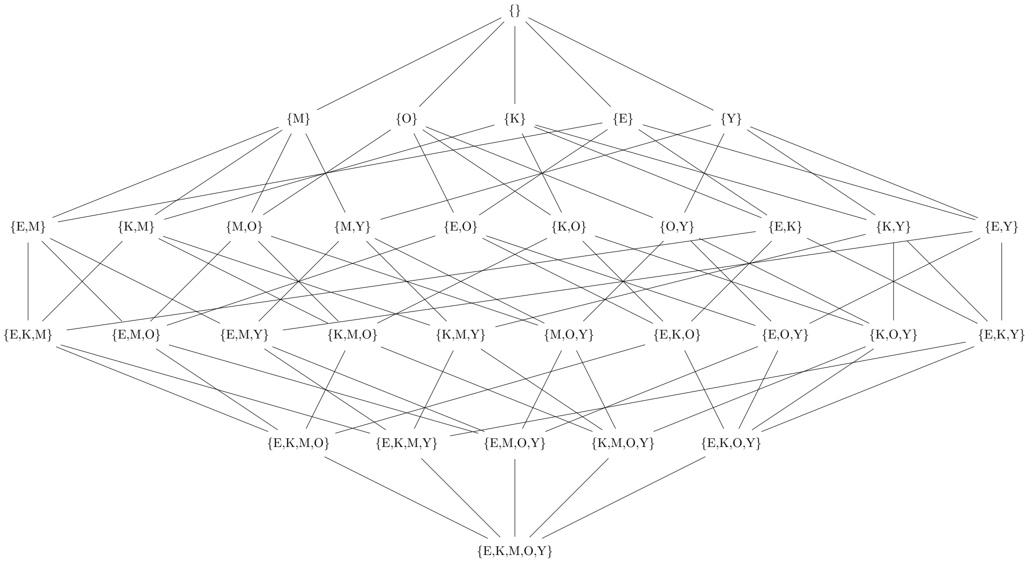
(a) Multiset of transactions T which subsets will characterized the possible frequent itemsets.

Item (I)	$\sigma(\cdot)$
M	3
O	3
N	2
K	5
E	4
Y	3
D	1
A	1
U	1
C	2
I	1

(b) $\mathbf{F_1}$ table representing the support σ count for each item $i \in I$

Item (I)	$\sigma(\cdot)$
M	3
O	3
K	5
E	4
Y	3

(c) Frequent Items with $\sigma(i) \geq 3$.



(d) Lattice $(\mathcal{L}_h, \subseteq)$ for all the possible frequent itemsets $|\mathcal{L}_h| = 2^{|\mathcal{L}_e|}$ generated from the items in $I = \mathcal{L}_e$. This kind of representation is also called Hasse Diagram (`https://en.wikipedia.org/wiki/Hasse_diagram`).

Figure 1: *A simple word-based use case scenario for generating frequent subwords showing the exponential nature of the Association Analysis problem.*

As per previous observations, the drawback of this algorithm is that we need to first generate all the possible itemset, and then return only the ones that satisfy the quality criterion. If we are interested in returning only the most general hypotheses matching the quality criterion, then we can provide a branch and bound approach[1] as the one provided in Algorithm 2 [3]: we can start scanning the lattice starting from the most general element $\top$ in the lattice (Line 2) by adding it in a *Queue* holding all the hypotheses that should be considered. Each hypothesis in the queue is added in the hypotheses set if it meets the quality requirement (Line 6) and, otherwise, we replace it in the queue with all of its specializations (Line 8). We can possibly use an heuristic (in the worst case scenario, $\mathcal{Q}$ itself) to reduce the number of hypotheses that we can return.

This pseudocode can be immediately implemented as a template class with virtual methods, where the definition of the quality assessment $\mathcal{Q}$, the hypothesis specialization and the prune operation are implemented for specific use case scenarios.

```
/**
 * Implementation of a generic general-to-specific computation with an heuristic pruning all the
     ↪ elements that do not satisfy the lift value
 *
 * @tparam DataType    Data Type over which we want to perform the element generation
 */
template <typename DataType> class GeneralToSpecificHeuristic {

protected:

    typedef std::priority_queue<std::pair<double, DataType>, // The data type that we are comparing
            std::vector<std::pair<double,DataType>> // The container of the data structure>
    >
            PriorityQueue;

    PriorityQueue queue;


    /**
     * Quality assessment predicate Q(h, D), where D is always the whole dataset. In particular, this
         ↪ quality assessment
     * function shall be a monotonic constraint.
     *
     * @param hypothesis  Hypothesis that we want to assess
     * @return            A boolean value
     */
    virtual bool quality_assessment(const std::pair<double, DataType>& hypothesis) const = 0;

    /**
     * Given an hypothesis, it provides a set of scored specializations
     *
     * @param hypothesis
     * @return
     */
    virtual std::vector<std::pair<double, DataType>> specialize_hypothesis(const DataType& hypothesis)
         ↪  const = 0;

    /**
     * Default method providing a pruning from the default queue
     *
     * @param queue   The queue will be updated accordingly to the predicate's specification
     */
    virtual void prune(PriorityQueue& queue) = 0;


public:
    GeneralToSpecificHeuristic() : queue{} {}

    std::unoredered_set<DataType> generate_hypotheses(const DataType& top, double score) {
        std::set<DataType> hypotheses;

        // C++'s PriorityQueue doesn't have a clear method: need to hardcode that
        queue = PriorityQueue{};
        // Insert the first element of the queue
        queue.emplace(score, top);
```

[1] See https://en.wikipedia.org/wiki/Branch_and_bound for more information on how to provide approximated solutions for NP-Complete problems using heuristic functions (in this case, $\mathcal{Q}$ is our heuristic).

```
        while (!queue.empty()) {
            const auto& h = queue.top(); // Passing the element by reference
            if (quality_assessment(h)) {
                hypotheses.insert(h.second);
                queue.pop();                // I'm done copying it: I can free the element
            } else {
                auto h_second = h.second; // Copy a part of the value...
                queue.pop();                // ... then, I can free it
                for (const auto& x : specialize_hypothesis(h_second)) {
                    queue.emplace(x);       // Re-populating the queue with the weighted specializations
                }
            }
            prune(queue);
        }

        return hypotheses;
    }

};
```

Given that we generally prefer generic rules to very specific rules [2], we can use this algorithm while generating the rules if $\mathcal{Q}$ is not necessarily anti-monotonic. On the other hand, for frequent itemsets generation we know that the min-support is anti-monotonic and we are also interested to the most specific itemset satisfying the min-support, and therefore we might expand only the hypotheses satisfying the quality criterion [3].

2.1 Frequent Itemsets Generation

Given a set of all the possible **transactions** T and a set of occuring **items** $I = \bigcup_{t \in T} t$, an itemset is an element of $i \in \wp(I)$. The **support** $\sigma(i)$ of any itemset i is defined as the number of transactions in T containing i as a subset. Given a minimum support threshold θ, an item i is frequent if its support is greater or equal than θ.

For the considerations from the introductory tutorial, we have that frequent itemsets generation resebles a binary classification via the indicator function, and so the condition $\mathcal{L}_h \simeq \wp(\mathcal{L}_e)$ holds. Consequently, we can generate a Partially Ordered Set[2] $(\mathcal{L}_h, \subseteq)$ that we can efficiently navigate and prune. Given that the original set I might produce an exorbitant amount of rules ($2^{|I|}-1 = 2047$), we might consider returning only the frequent itemsets generated from the most frequent items (Figure 1c). Therefore, we need to introduce a minimum support threshold θ allowing us to reduce the size of $\mathcal{L}_e$, and therefore the size of $\mathcal{L}_h$. This approach reduces the size of the elements that we want to visit, but does not prevent us from visiting all the hypothesis in $\mathcal{L}_h$ using the same brute force approach presented in Algorithm 1: in particular, we need to determine the support count for every candidate itemset by comparing each candidate in I against every transaction in T. This approach is not only inefficient because we generate more results than the one we are returning, but also because this approach would require multiple scans to T, thus providing an inefficient solution in terms of I/O frequency access.

Given that in the former tutorial the min-support condition is an anti-monotonic condition, we can visit the tree starting from the most general pattern towards the most specific item: if we exclude the empty set, this imply that we need to visit the lattice from the singleton sets generated from I. Given this initial observation, we can derive a first and an easy implementation of such algorithm, which is the **Apriori Algorithm**: for each i-th iteration step from $i = 1$ to $|I|$, prune $\mathbf{F_i}$ using a min-support θ so to contain only p_i^θ frequent patterns of size i, then compute $\mathbf{F_{i+1}} := \mathbf{F_1} \times \mathbf{F_i}$; you can also stop the iteration at the j-th step if the resulting $\mathbf{F_j}$ is empty. At this point, we can see [4] that we are reducing the number of the overall generated candidates from $2^j - 1$ down to $\sum_{1 \leq i \leq j} \binom{p_i^\theta}{i} \leq 2^j - 1$: we can also prove that these two quantities are going to be the same if and only if the pruning is ineffective, and therefore does not decrease the overall number of the generated patterns. Even though we could further optimize this approach (see Exercise 1), this cross-product approach is still generating more candidates than the ones that we are required. In fact, at each stage we still need to prune $\mathbf{F_i}$ so to reduce the number of all the possible combinations down to $p_i^\theta \cdot p_1^\theta$.

[2] See `https://en.wikipedia.org/wiki/Partially_ordered_set` if you want to know more on partially ordered sets and have a more formal definition.

In order to further reduce the time required to generate answers, the **FPGrowth** algorithm adopts a completely different kind of strategy: instead of generating intermediate results that then need to be removed (see the previous algorithm), I can directly extend an ad hoc representation of T so to naturally embed the value of σ associated to each potential frequent itemset that can be generated by T. Furthermore, I can generate only the *candidates* that are frequent by representing T as a prefix tree, so that each node represents one item in I alongside with its support count, and each path in the tree will represent a candidate, which frequency is determined by the terminal node in the path.

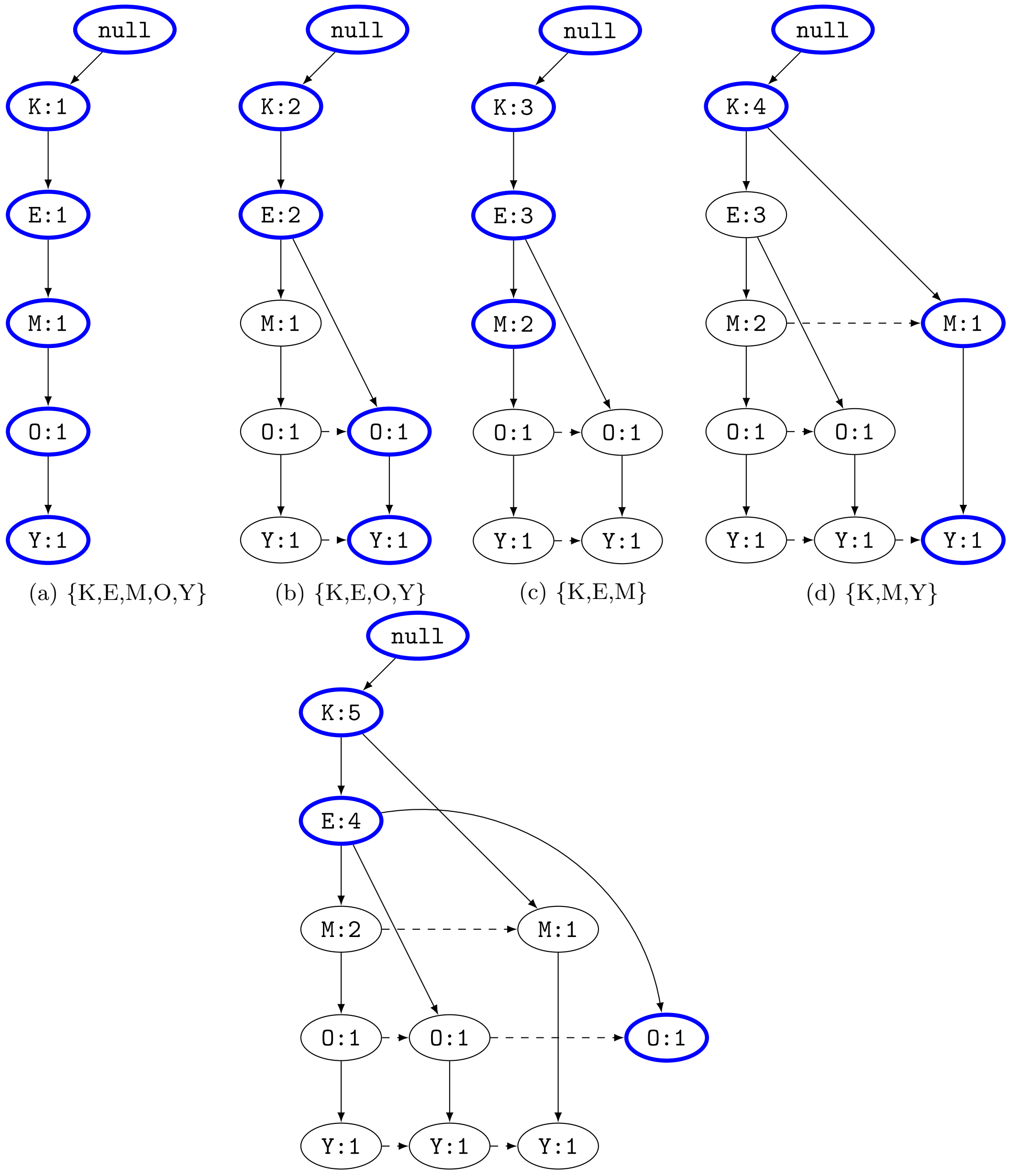
First, given the set of transitions T, I can compute σ as a map implementing the finite function $\sigma\colon I \to \mathbb{N}$ (Line 55, Figure 1b), then I can prune from I and hence from σ all the infrequent candidates having a support less than the minimum (Line 62, Figure 1c). Then, we can use a set to order all the elements in the pruned $\mathbf{F_1}$ by decreasing support value (Line 68) via a `frequency_comparator` class: we want to sort each support-item pair by decreasing support and then by decreasing alphabetical order. This is implemented by converting pairs into tuples via `tie`, over which a lexicographical order is defined [3]. Then, I can simply sort each transaction `transaction` by `frequency_comparator` by just simply scanning each frequent item (Line 81) and then checking if that item is contained in each transaction (Line 85). At the very beginning, the FPTree consists on one single root node representing the empty pattern (Line 76): while visiting each transaction $t \in T$ starting from the item i with highest support, we check if there is a child with the same name: if this node already exist, then increment the support value stored in the current node child `curr_fpnode_child` (Line 116) and set that node as the next one to be visited (Line 119). Otherwise, if no such node exists, then we need to generate a new child node to the current node (Line 95); in order to efficiently navigate across all the FPNodes representing the same item belonging to different possible subpaths, then we need to extend the σ table with a pointer to the first inserted node for this item (Line 107) and to connect all the previously inserted nodes for the same item in a linked list fashion (Line 104).

After ordering each transaction t in the database T by descending order of σ and by removing the infrequent candidates (e.g., $\{M, O, N, K, E, Y\} \mapsto \{K, E, M, O, Y\}$ updating the FPTree to Figure 2a), we will incrementally add such sets to generate the final FPTree (Figure 2e after the insertion of the last set $\{K, E, O\}$). Such tree will be then associated to a updated frequent items table (Figure 1c), where each item will point to the leftmost occurrence of the item in the FPTree. The size of an FPTree is typically smaller than the size of T because many transaction in the database often share a few items in common, even though the physical storage of such tree is higher than T itself because we need to introduce additional pointers: as an example, the $\{K, E, M\}$ (Figure 2c) transaction is also included in the $\{K, E, M, O, Y\}$ one because $\{K, E, M, O, Y\}$ is actually a superset of $\{K, E, M\}$.

However, such representation does not guarantee that the FPTree representation is I-order invariant. This suggests that we would like to determine in advance, which I permutation ensures an optimal compression of the data structure. However, at the time of the writing, this is not possible, as long we would need to generate first all the trees for each possible permutation of I, thus reducing us to solve a factorial problem. In these cases, it is preferred to order I using a heuristic that takes into account the following observations of the average case: ordering the items from the lowest to the highest support provides a denser tree, while doing the opposite provides a better compression.

```
 1 #ifndef TUTORIALS_FPTREE_H
 2 #define TUTORIALS_FPTREE_H
 3
 4 #include <cstdint>
 5 #include <map>
 6 #include <memory>
 7 #include <set>
 8 #include <string>
 9 #include <vector>
10 #include <utility>
11 #include <algorithm>
12 #include <iostream>
13
14 #define ForAll(T)      template <typename T>
15 #define ForAll2(T,K)   template <typename T, typename K>
16 #define ForAll3(T,K,U)  template <typename T, typename K,typename U>
```

[3] "A lexicographical comparison is the kind of comparison generally used to sort words alphabetically in dictionaries" See `http://www.cplusplus.com/reference/algorithm/lexicographical_compare/` for more details on how lexicographical order was implemented in C++ over sequential data structures. See the Appendix for a custom implementation of such order.



(a) {K,E,M,O,Y}

(b) {K,E,O,Y}

(c) {K,E,M}

(d) {K,M,Y}

(e) {K,E,O}

Figure 2: Figure providing the incremental update steps generating the FPTree from the multiset T in Figure 1a. Items belonging to different transactions are represented as different FPNodes connected by a dashed line, which sum corresponds to the one provided by σ.

```

ForAll(Item) using Transaction       = std::vector<Item>;
ForAll(Item) using TransformedPrefixPath = std::pair<std::vector<Item>, uint64_t>;
ForAll(Item) using Pattern           = std::pair<std::set<Item>, uint64_t>;

ForAll(Item) struct FPNode {
    const Item                       item;
    uint64_t                         frequency;
    std::shared_ptr<FPNode<Item>>    node_link;
    std::weak_ptr<FPNode<Item>>      parent;
    std::vector<std::shared_ptr<FPNode>> children;

    FPNode(const Item& item, const std::shared_ptr<FPNode<Item>>& parent) :
            item( item ), frequency( 1 ), node_link( nullptr ), parent( parent ), children() {
    }
};

ForAll(Item) struct FPTree {
    std::shared_ptr<FPNode<Item>>            root;
    std::map<Item, std::shared_ptr<FPNode<Item>>> header_table;
    uint64_t                                 minimum_support_threshold;

    // order items by decreasing frequency
    struct frequency_comparator
    {
        bool operator()(const std::pair<Item, uint64_t> &lhs, const std::pair<Item, uint64_t> &rhs)
            ↪ const
        {
            return std::tie(lhs.second, lhs.first) > std::tie(rhs.second, rhs.first);
        }
    };

    FPTree(const std::vector<Transaction<Item>>& transactions, uint64_t minimum_support_threshold) :
            root(std::make_shared<FPNode<Item>>( Item{}, nullptr )), header_table(),
            minimum_support_threshold( minimum_support_threshold ) {

        std::cout << "initialization..." << std::endl;

        // scan the transactions counting the frequency of each item
        std::map<Item, uint64_t> frequency_by_item;
        for ( const Transaction<Item>& transaction : transactions ) {
            for ( const Item& item : transaction ) {
                ++frequency_by_item[item];
            }
        }
        // keep only items which have a frequency greater or equal than the minimum support threshold
        for ( auto it = frequency_by_item.cbegin(); it != frequency_by_item.cend(); ) {
            const uint64_t item_frequency = (*it).second;
            if ( item_frequency < minimum_support_threshold ) { frequency_by_item.erase( it++ ); }
            else { ++it; }
        }

        std::set<std::pair<Item, uint64_t>, frequency_comparator> items_ordered_by_frequency(
            ↪ frequency_by_item.cbegin(), frequency_by_item.cend());

        // start tree construction
        std::cout << "starting␣to␣construct␣the␣tree." << std::endl;

        // scan the transactions again
        size_t i = 0;
        for ( const Transaction<Item>& transaction : transactions ) {
            auto curr_fpnode = root;
            if (!(i % 1000)) std::cout << "Transaction␣#"<< i << std::endl;
            i++;

            // select and sort the frequent items in transaction according to the order of
                ↪ items_ordered_by_frequency
            for ( const auto& pair : items_ordered_by_frequency ) {
                const Item& item = pair.first;

                // check if item is contained in the current transaction
                if ( std::find( transaction.cbegin(), transaction.cend(), item ) != transaction.cend()
                    ↪ ) {
```

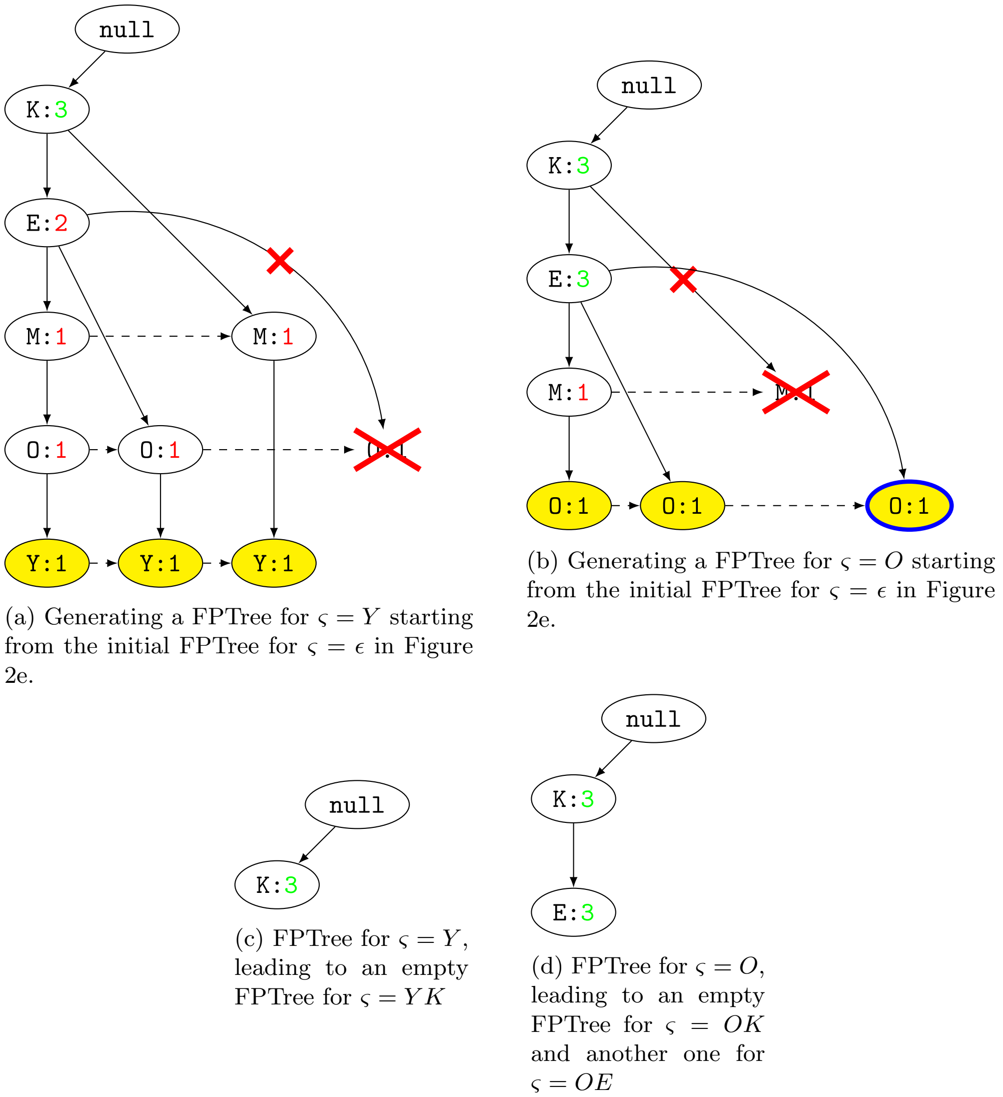
```
                    // insert item in the tree

                    // check if curr_fpnode has a child curr_fpnode_child such that curr_fpnode_child.
                         ↪ item = item
                    const auto it = std::find_if(
                            curr_fpnode->children.cbegin(), curr_fpnode->children.cend(), [item](const
                                 ↪ std::shared_ptr<FPNode<Item>>& fpnode) {
                                return fpnode->item == item;
                            } );
                    if ( it == curr_fpnode->children.cend() ) {
                        // the child doesn't exist, create a new node
                        const auto curr_fpnode_new_child = std::make_shared<FPNode<Item>>( item,
                             ↪ curr_fpnode );

                        // add the new node to the tree
                        curr_fpnode->children.push_back( curr_fpnode_new_child );

                        // update the node-link structure
                        if ( header_table.count( curr_fpnode_new_child->item ) ) {
                            auto prev_fpnode = header_table[curr_fpnode_new_child->item];
                            while ( prev_fpnode->node_link ) { prev_fpnode = prev_fpnode->node_link; }
                            prev_fpnode->node_link = curr_fpnode_new_child;
                        }
                        else {
                            header_table[curr_fpnode_new_child->item] = curr_fpnode_new_child;
                        }

                        // advance to the next node of the current transaction
                        curr_fpnode = curr_fpnode_new_child;
                    }
                    else {
                        // the child exist, increment its frequency
                        auto curr_fpnode_child = *it;
                        ++curr_fpnode_child->frequency;

                        // advance to the next node of the current transaction
                        curr_fpnode = curr_fpnode_child;
                    }
                }
            }
        }
    }

    bool empty() const {
        assert( root );
        return root->children.size() == 0;
    }
};
```

At this stage, we want to generate all the possible frequent itemsets with a min-support θ. After observing that prefixes of a given length k induce equivalence classes composed of distinct itemsets[4], we can generate all the possible frequent itemsets using a Divide and Conquer approach[5]: from a single FPTree associated to a initial prefix ς of length k (initially $k = 0$ and ς is the empty string ϵ), we can potentially generate as frequent itemsets all the ςi-s candidate where $i \in \mathrm{dom}(\sigma)$. Before updating σ and I for the new FPTree associated to ςi, we need to restructure the FPTree associated to ς as follows: first, prune the FPTree so that its only leaves will be the nodes containing the item i, then, replace each non-leaf node $x : s$ support count s as the sum of the supports of all the descendant leaves and, as a consequence, update $\sigma(x)$ by summing up the support of all the x-labelled nodes in the FPTree; last, before the recursive call, prune out all the nodes in the FPTree and in I and σ that have a support which is less than θ.

Going back to our simple word example, Figure 2e represents the FPTree for $\varsigma = \epsilon$ over which we're going to iterate over all the frequent itemsets of size 1 in I (or $\mathrm{dom}(\sigma)$). For demonstration

[4] Using a simplistic definition, $[p_1 p_2 \dots p_k] = \{t \in T | p_1 p_2 \dots p_k \subseteq t\}$ where $p_1 p_2 \dots p_k$ is a prefix of length k. More formally, we can define a equivalence relation $\sim_k$ over two itemsets i and j such that $i \sim_k j$ holds if and only if the first k elements of the `frequency_comparator`-ordered sets are the same. This similarity function induces the definition of an equivalence class $[p_1 p_2 \dots p_k] = \{\, t \in T \mid t \sim_k p_1 p_2 \dots p_k \,\}$. See `https://en.wikipedia.org/wiki/Equivalence_class` for more information on equivalence classes.

[5] See §4 from the *Bachmann–Landau notation* workshop.



(a) Generating a FPTree for $\varsigma = Y$ starting from the initial FPTree for $\varsigma = \epsilon$ in Figure 2e.

(b) Generating a FPTree for $\varsigma = O$ starting from the initial FPTree for $\varsigma = \epsilon$ in Figure 2e.

(c) FPTree for $\varsigma = Y$, leading to an empty FPTree for $\varsigma = YK$

(d) FPTree for $\varsigma = O$, leading to an empty FPTree for $\varsigma = OK$ and another one for $\varsigma = OE$

Figure 3: FPGrowth algorithm

purposes, we will only consider $Y \in I$ and $O \in I$.

Figure 3a shows the restructuring steps required for transforming the initial FPTree for $\varsigma = \epsilon$ into the FPTree for $\varsigma = Y$: we need to remove the rightmost $O : 1$ node because it is associated to no leaf, and we consequently remove the edge with `E : 2`. At the end of the road, Figure 3c is the resulting restructured FPTree which, after another recursive call, it will only generate the itemset $\{Y, K\}$. Similarly, Figure 3b showing the pruning algorithm running when the leaf nodes for O are selected: we will obtain another FPTree (Figure 3d), where both itemsets $\{O, K\}$ and $\{O, E\}$ might be generated.

As we might observe from the former examples, the restructuring operation of the FPTree at each recursive step boils down to generating all the paths (by scanning the item single linked list as in Line 85) from each node representing an item $i \in I$ in a prefix path (Line 52) towards the root (the only node having an empty parent, Line 69), which support will be equal to the support of the current instance of item i (Line 71). Then, each path will be converted into as many equivalent transactions as the support associated to it (Line 89), over which we will create an FPTree for ςi (Line 103) pruning the unfrequent items. The Divide and Conquer strategy guarantees that a recursive call over ςi will generate all the potential frequent patterns having ςi as a prefix: therefore, we need to both extend such generated patterns with i (Line 123), and to add the singleton $\{i\}$ (Line 120) with frequency $\sigma(i)$ (Line 115) to the set of patterns to be returned by the current function call. If the FPTree contains just a single path (e.g., Figure 3c and Figure 3d), then we can directly generate all the possible $2^k - 1$ itemsets by scanning the path only once (Line 14). By doing so, we avoid the pruning and the repair costs that we are facing while we're using the Divide and Conquer approach over all the items in I: given that the leaf element is already frequent, then all the patterns associated to the given prefix will be also frequent by construction.

```
ForAll(Item) bool contains_single_path(const std::shared_ptr<FPNode<Item>>& fpnode) {
    assert( fpnode );
    if ( fpnode->children.size() == 0 ) { return true; }
    if ( fpnode->children.size() > 1 ) { return false; }
    return contains_single_path( fpnode->children.front() );
}

ForAll(Item) bool contains_single_path(const FPTree<Item>& fptree) {
    return fptree.empty() || contains_single_path( fptree.root );
}

ForAll(Item) std::set<Pattern<Item>> fptree_growth(const FPTree<Item>& fptree) {
    if ( fptree.empty() ) { return {}; }
    if ( contains_single_path( fptree ) ) {
        // generate all possible combinations of the items in the tree
        std::set<Pattern<Item>> single_path_patterns;

        // for each node in the tree
        assert( fptree.root->children.size() == 1 );
        auto curr_fpnode = fptree.root->children.front();
        while ( curr_fpnode ) {
            const Item& curr_fpnode_item = curr_fpnode->item;
            const uint64_t curr_fpnode_frequency = curr_fpnode->frequency;

            // add a pattern formed only by the item of the current node
            Pattern<Item> new_pattern{ { curr_fpnode_item }, curr_fpnode_frequency };
            single_path_patterns.insert( new_pattern );

            // create a new pattern by adding the item of the current node to each pattern generated
                ↪ until now
            for ( const Pattern<Item>& pattern : single_path_patterns ) {
                Pattern<Item> new_pattern{ pattern };
                new_pattern.first.insert( curr_fpnode_item );
                assert( curr_fpnode_frequency <= pattern.second );
                new_pattern.second = curr_fpnode_frequency;

                single_path_patterns.insert( new_pattern );
            }

            // advance to the next node until the end of the tree
            assert( curr_fpnode->children.size() <= 1 );
            if ( curr_fpnode->children.size() == 1 ) { curr_fpnode = curr_fpnode->children.front(); }
            else { curr_fpnode = nullptr; }
        }

        return single_path_patterns;
    } else {
        // generate conditional fptrees for each different item in the fptree, then join the results

        std::set<Pattern<Item>> multi_path_patterns;

        // for each item in the fptree
        for ( const auto& pair : fptree.header_table ) {
            const Item& curr_item = pair.first;

            // build the conditional fptree relative to the current item
            // start by generating the conditional pattern base
            std::vector<TransformedPrefixPath<Item>> conditional_pattern_base;

            // for each path in the header_table (relative to the current item)
            auto path_starting_fpnode = pair.second;
            while ( path_starting_fpnode ) {
                // construct the transformed prefix path

                // each item in th transformed prefix path has the same frequency (the frequency of
                    ↪ path_starting_fpnode)
                const uint64_t path_starting_fpnode_frequency = path_starting_fpnode->frequency;

                auto curr_path_fpnode = path_starting_fpnode->parent.lock();
                // check if curr_path_fpnode is already the root of the fptree
                if ( curr_path_fpnode->parent.lock() ) {
                    // the path has at least one node (excluding the starting node and the root)
                    TransformedPrefixPath<Item> transformed_prefix_path{ {},
                         ↪ path_starting_fpnode_frequency };

                    while ( curr_path_fpnode->parent.lock() ) {
                        assert( curr_path_fpnode->frequency >= path_starting_fpnode_frequency );
                        transformed_prefix_path.first.push_back( curr_path_fpnode->item );

                        // advance to the next node in the path
                        curr_path_fpnode = curr_path_fpnode->parent.lock();
                    }

                    conditional_pattern_base.push_back( transformed_prefix_path );
                }

                // advance to the next path
                path_starting_fpnode = path_starting_fpnode->node_link;
            }

            // generate the transactions that represent the conditional pattern base
            std::vector<Transaction<Item>> conditional_fptree_transactions;
            for ( const TransformedPrefixPath<Item>& transformed_prefix_path : conditional_pattern_base
                 ↪  ) {
                const std::vector<Item>& transformed_prefix_path_items = transformed_prefix_path.first;
                const uint64_t transformed_prefix_path_items_frequency = transformed_prefix_path.second
                     ↪ ;

                Transaction<Item> transaction = transformed_prefix_path_items;

                // add the same transaction transformed_prefix_path_items_frequency times
                for ( auto i = 0; i < transformed_prefix_path_items_frequency; ++i ) {
                    conditional_fptree_transactions.push_back( transaction );
                }
            }

            // build the conditional fptree relative to the current item with the transactions just
                 ↪ generated
            const FPTree<Item> conditional_fptree( conditional_fptree_transactions, fptree.
                 ↪ minimum_support_threshold );
            // call recursively fptree_growth on the conditional fptree (empty fptree: no patterns)
            std::set<Pattern<Item>> conditional_patterns = fptree_growth( conditional_fptree );

            // construct patterns relative to the current item using both the current item and the
                 ↪ conditional patterns
            std::set<Pattern<Item>> curr_item_patterns;

            // the first pattern is made only by the current item
            // compute the frequency of this pattern by summing the frequency of the nodes which have
                 ↪ the same item (follow the node links)
            uint64_t curr_item_frequency = 0;
            auto fpnode = pair.second;
            while ( fpnode ) {
                curr_item_frequency += fpnode->frequency;
                fpnode = fpnode->node_link;
            }
            // add the pattern as a result
            Pattern<Item> pattern{ { curr_item }, curr_item_frequency };
            curr_item_patterns.insert( pattern );

            // the next patterns are generated by adding the current item to each conditional pattern
            for ( const Pattern<Item>& pattern : conditional_patterns ) {
                Pattern<Item> new_pattern{ pattern };
                new_pattern.first.insert( curr_item );
                assert( curr_item_frequency >= pattern.second );
                new_pattern.second = pattern.second;

                curr_item_patterns.insert( { new_pattern } );
            }

            // join the patterns generated by the current item with all the other items of the fptree
            multi_path_patterns.insert( curr_item_patterns.cbegin(), curr_item_patterns.cend() );
```

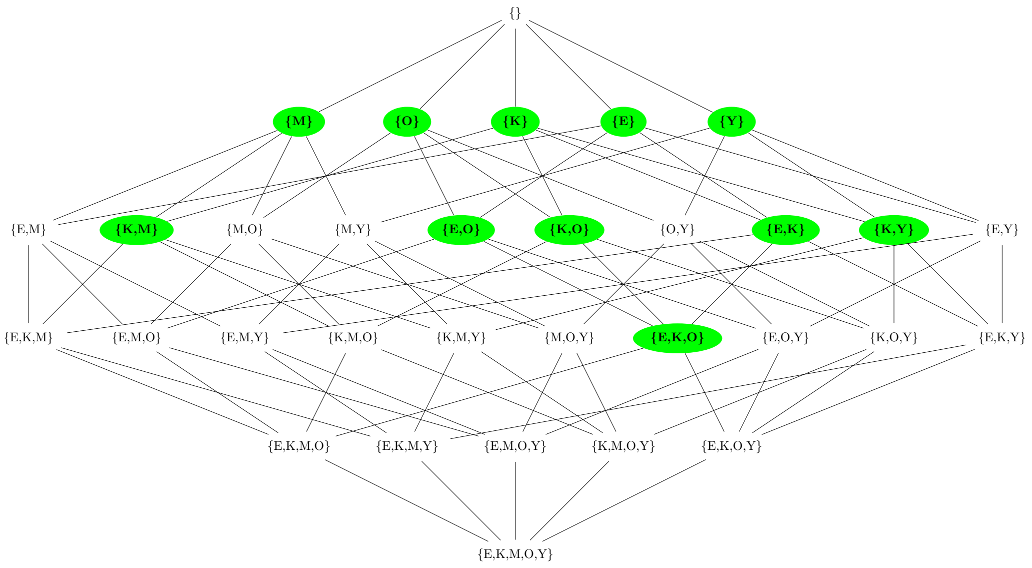


Figure 4: The itemsets in green remark which are the elements that are returned by the FPGrowth function using a *min-support* of θ.

```
        }

        return multi_path_patterns;
    }
}

#endif // FPTREE_HPP
```

Figure 4 highlights the elements returned by running the FPGrowth algorithm returning all the frequent patterns returned from the database T when represented as the FPTree in Figure 2e. As we might notice in the algorithm, the only computational cost that we have to pay at each recursive call is the repair of the FPTree. As you might also observe from comparing the tree from two different recursive calls (Figure 3a vs. Figure 3b), we will have that this algorithm does not provide subsequent recursion calls with a perfectly balanced trees, as the resulting recurrence relation resembles a *Homogeneous linear recurrence relation with constant coefficients*. So, albeit the Divide and Conquer approach allows to easily parallelise the algorithm (see Exercise 2d), such approach might boils down into an inefficient implementation in the worst case scenario. Nevertheless, we can show that this implementation is always faster than the Apriori approach in the average case scenario, thus resulting in a more efficient algorithm for real world datasets.

2.2 Rule Generation

A rule or **association pattern** $X \to Y$ might be interpreted as "*if X occurs in the current transaction $t \in T$, then it is likely that Y will occur, too*": we can call X as **antecedent** (or head) and Y as a **consequent** (or tail). Each frequent itemset Z can be considered as a rule $\emptyset \to Z$ stating that Z is a possible configuration within our dataset, while $Z \to \emptyset$ implies that no (further) suggestion can be given from Z. As a result, we might ignore all the rules having either empty antecedent or empty consequent as they are not interesting.

Any rule can be represented by the following structure, which just contains two vectors for both the antecedent and the consequent:

```
/**
 * Represents a simple mined rule
 */
struct Rule {
    std::vector<std::string> head; ///< Represents the head of the rule
    std::vector<std::string> tail; ///< Represents the tail of the rule
    Rule() = default;
    Rule(const Rule& rule) = default;
    Rule& operator=(const Rule& rule) = default;

    /**
     * Creating a rule from a vector having the head filled and the tail empty
     * @param itemset
     */
    Rule(std::vector<std::string>& itemset);

    /**
     * Creating a rule from a pattern having the head filled and the tail empty
     * @param itemset
     */
    Rule(const Pattern<std::string>& itemset);

    /**
     * Printing some rules in output
     * @param os
     * @param rule
     * @return
     */
    friend std::ostream &operator<<(std::ostream &os, const Rule &rule);

    /**
     * Comparison operator: using the formal definition of precedence!
     */
    bool operator<(const Rule &rhs) const;
    bool operator>(const Rule &rhs) const;
    bool operator<=(const Rule &rhs) const;
    bool operator>=(const Rule &rhs) const;
    bool operator==(const Rule &rhs) const;
    bool operator!=(const Rule &rhs) const;
};
```

In particular, we can force the non-default constructors to generate a rule $\emptyset \to Z$ from a given frequent itemset Z as follows:

```
Rule::Rule(std::vector<std::string> &itemset) : head{itemset}, tail{} {}

Rule::Rule(const Pattern<std::string> &itemset) : head{itemset.first.size()} {
    // Copying the set's elements inside the predecessor
    std::copy(itemset.first.begin(), itemset.first.end(), head.begin());
}

std::ostream &operator<<(std::ostream &os, const Rule &rule) {
    os << "{";
    for (size_t i = 0, N = rule.head.size(); i < N; i++) {
        os << rule.head[i];
        if (i != (N-1)) os << ",␣";
    }
    os << "}␣=>␣{";
    for (size_t i = 0, N = rule.tail.size(); i < N; i++) {
        os << rule.tail[i];
        if (i != (N-1)) os << ",␣";
    }
    os << "}";
    return os;
}
```

Last, in order to insert the Rule in a `unordered_set`, we also need to create a `hash` template class. One possible hashing function is the following:

```
namespace std {
    template<>
    class hash<Rule> {
    public:
        size_t operator()(const Rule &s) const
        {
            size_t i = 31;
            size_t j = 7;
            for (const std::string& x : s.head) {
                i = std::hash<std::string>()(x) * 31 + i;
            }
            for (const std::string& x : s.tail) {
                j = std::hash<std::string>()(x) * 7 + i;
            }
            return i ^ ( j << 1 );
        }
    };
}
```

While the partial order for subsets is trivial and matches with the definition of the subset/supset predicate, the definition of such a predicate is non trivial for rules (see also the introductory tutorial to *In-Silico* Learning). Given that the former potentially provides more predictions than the latter,[6] we can refer to $\emptyset \rightarrow Z$ as our first or **top** element $\top$ of our lattice and $Z \rightarrow \emptyset$ as the last or **bottom** $\bot$. We can observe that a generalization of two rules $X \rightarrow Y$ and $T \rightarrow U$ can be defined as a rule $X \cap T \rightarrow Y \cup U$, from which the following generalization predicate follows (Lemma 1 in the Appendix):

$$T \rightarrow U \preceq X \rightarrow Y \Leftrightarrow T \subseteq X \wedge U \supseteq Y$$

This precedence formal definition can be now immediately implemented for Rules after implementing the $\supseteq$ for vectors (`IsSupsetOf`):

```
/**
 * Subset relation among two vectors
 *
 * @tparam T   Types of the values associated to the vector
 * @param A    Left vector, the supposed superset
 * @param B    Right vector, the supposed subset
 * @return     Whether B is a subset of or equal to A
 */
template <typename T> bool IsSupsetOf(std::vector<T> A, std::vector<T> B) {
    std::sort(A.begin(), A.end());
    std::sort(B.begin(), B.end());
    return std::includes(A.begin(), A.end(), B.begin(), B.end());
}


bool Rule::operator<(const Rule &rhs) const {
    return (IsSupsetOf(tail, rhs.tail) && IsSupsetOf(rhs.head, head));
}

bool Rule::operator>(const Rule &rhs) const {
    return rhs < *this;
}

bool Rule::operator<=(const Rule &rhs) const {
    return (*this < rhs) || (*this == rhs);
}

bool Rule::operator>=(const Rule &rhs) const {
    return (*this > rhs) || (*this == rhs);
}

bool Rule::operator==(const Rule &rhs) const {
    return IsSupsetOf(head,rhs.head) && IsSupsetOf(rhs.head,head) &&
            IsSupsetOf(tail,rhs.tail) && IsSupsetOf(rhs.tail,tail);
}

bool Rule::operator!=(const Rule &rhs) const {
    return !(rhs == *this);
}
```

At this point, we have the complete characterization of what a Rule composed of items as strings should be. Now, we can also start to implement a `RulesFromFrequentItemset` class inheriting from `GeneralToSpecificHeuristic`. As a first step, we need to implement the specialization method: we can immediately characterize such method by using the previous formal definition of $\preceq$:

[6] Please observe that $X \rightarrow Y$ cannot be properly interpreted as a logical implication: given that $Y = Z\backslash X$, we cannot say that $\forall x. x \in X \Rightarrow x \in Z\backslash X$, and therefore we cannot use a more formal setting to draw conclusions on what a precedence relation $\preceq$ and a generalization operator *mgg* should be.

```
std::vector<std::pair<double, Rule>> RulesFromFrequentItemset::specialize_hypothesis(const Rule &
    ↪ hypothesis) const {
    std::vector<std::pair<double, Rule>> result;
    for (const std::string& x : hypothesis.head) { // For each x appearing in the head of a hypothesis
        ↪ ...
        Rule newRule;                            // ... generate a novel rule ...
        {
            // ... having as a tail the tail of hypothesis where x is added ...
            std::vector<std::string> succ{hypothesis.tail};
            succ.emplace_back(x);
            // (Please Note: the fastest approach of removing duplicates from a vector is to create a
                ↪ set, and then move it back to a vector)
            {
                std::set<std::string> setSucc( succ.begin(), succ.end() );
                newRule.tail.assign(setSucc.begin(), setSucc.end() );
            }
        }

        // ... and as a head all the elements belonging to the hypothesis's head minus x, that was
            ↪ currently added to the tail
        for (const std::string& y : hypothesis.head)
            if (x != y) newRule.head.emplace_back(y);

        // ... Associate a score (e.g., lift) to the newly generated rule
        result.emplace_back(scorer.lift(newRule), newRule);
    }
    // Return all the ranked specializations of the hypothesis
    return result;
}
```

At this stage, we only need to implement a quality criterion for the rules $\mathcal{Q}$ in order to prune the non-interesting ones. We could evaluate any rule $X \to Y$ mainly via these three scoring functions:

- (rule) **support**: it remarks how many times a rule appears in the dataset as a itemset (*high support values mean should apply to a large amount of cases*):

$$s(X \to Y) = \frac{\sigma(X \cup Y)}{|T|}$$

- **confidence**: it remarks the frequency of Y appearing in T when also X does (*high confidence values mean that the rule should be often correct*):

$$c(X \to Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

- **lift**: compares the rule confidence with the null hypothesis's confidence (*high lift values indicate that the rule is not just a coincidence*):

$$\ell(X \to Y) = \frac{\sigma(X \cup Y)}{\sigma(X)\sigma(Y)}$$

We could observe that the *lift* measure is similar to the covariance coefficient that we described in the *Machine Learning* tutorial: in fact, this measure will return 1 if the support for the itemset generated from the rule is as frequent as the product of the antecedent's and the consequent's support, thus suggesting that antecedent and consequent are statistically independent. This implies that all the interesting rules will be the ones having a lift greater than one. After recognizing that all these definitions require a definition of σ as defined by the FPGrowth algorithm returning a set of patterns:

```
// <header file>
/**
 * This struct stores the association itemset/number_of_occurrences returned from the FPGrowth
      ↪ algorithm and returns
 * a set of data mining metrics
 */
struct DataMiningMetrics {
    std::map<std::vector<std::string>, unsigned long, VTLexic> f;///< Storing the item-support
         ↪ information from the FPGrowth algorithm
    double sumAll = 0.0;                                        ///<  Sum of all the supports for |T|

    /**
     * Initialization via the output of the FPGrowth algorithm
     * @param S
     */
    DataMiningMetrics(const std::set<Pattern<std::string>>& S);

    /**
     * Support definition for each item
     * @param i
     * @return
     */
    size_t support(const std::vector<std::string>& i) const;

    /**
     * Rule support
     * @param r
     * @return
     */
    double support(const Rule& r) const;

    /**
     * Rule lift
     * @param r
     * @return
     */
    double lift(const Rule& r) const;
};




// <cpp file>
DataMiningMetrics::DataMiningMetrics(const std::set<Pattern<std::string>> &S) {
    for (auto x : S) { // For each pattern
        std::vector<std::string> v{}; // Store it as a vector
        v.reserve(x.first.size());
        for (auto it = x.first.begin(); it != x.first.end(); ) {
            v.push_back(std::move(x.first.extract(it++).value()));
        }
        f[v] = x.second; // Associate the support to the rule represented as a vector
        sumAll += x.second; // Increment the number of the (frequent) transactions (Suggestion: replace
             ↪  it with the actual size of |T|)
    }
}

size_t DataMiningMetrics::support(const std::vector<std::string> &i) const {
    size_t sum = 0;
    for (auto it = f.begin(); it != f.end(); it++) {
        if (IsSupsetOf(it->first, i)) {
            sum += it->second;
        }
    }
    return sum;
}

double DataMiningMetrics::support(const Rule &r) const {
    std::vector<std::string> unione;
    for (const std::string& x: r.head) unione.emplace_back(x);
    for (const std::string& x: r.tail) unione.emplace_back(x);
    std::sort(unione.begin(), unione.end());
    unione.erase(std::unique(unione.begin(), unione.end()), unione.end());
    return ((double)support(unione)) / sumAll;
}

double DataMiningMetrics::lift(const Rule &r) const {
    std::vector<std::string> unione;
    for (const std::string& x: r.head) unione.emplace_back(x);
    for (const std::string& x: r.tail) unione.emplace_back(x);
    std::sort(unione.begin(), unione.end());
    unione.erase(std::unique(unione.begin(), unione.end()), unione.end());
    double pup = ((double)support(unione))/sumAll;
    double ppred = ((double)support(r.head) / sumAll);
    double psucc = ((double)support(r.tail) / sumAll);
    double final = (pup)/(ppred*psucc);
    return final;
}
```

As we might observe from the class's constructor, the support of all the non-frequent patterns will be considered as zero, as such pattern will not be stored inside the `f` map.

After proving that the minimum lift rule is anti-monotonic (see Lemma 2 in the Appendix), then we can directly use the Branch-and-Bound implementation using the `lift(r) > 1` as a quality criterion:

```
bool RulesFromFrequentItemset::quality_assessment(const std::pair<double, Rule> &hypothesis) const {
    return hypothesis.first > 1; // Getting only the plausible elements
}
```

The same predicate is also going to be used for the `prune` method: please observe that C++ does not allow to directly scan the elements contained in a queue and then to prune those, and therefore we are forced to dump the element satisfying the quality criterion into a vector `v` first, and then inserting all the elements in `v` back to the queue:

```
void RulesFromFrequentItemset::prune(GeneralToSpecificHeuristic<Rule>::PriorityQueue &queue) {
    /*
     * Priority queues do not allow to directly remove the elements: therefore, I need to always
         ↪ remove the elements
     * from the queue, and then emplace the element in v if it meets the lift requirements
     */
    std::vector<std::pair<double, Rule>> v;
    while (!queue.empty()) {
        auto qt = queue.top();
        if ((scorer.lift(qt.second)) > 1) {
            v.emplace_back(qt);                 // Preserving the element
        }
        queue.pop();                            // Always removing the element from the queue
    }
    auto it = v.begin();                        // Iterating over the vector, while freeing the memory
    while (it != v.end()) {
        queue.push(*it);                        // Re-adding the preserved element back
        it = v.erase(it);                       // Clearing the memory from the vector.
    }
}
```

In this scenario, we then need to feed a reference of `DataMiningMetrics` to the `RulesFromFrequentItemset` constructor:

```
RulesFromFrequentItemset::RulesFromFrequentItemset(DataMiningMetrics &scorer) : scorer(scorer) {
}
```

Last, we can define a `generate_hypotheses` method generating a $\top$ rule from the frequent pattern, and then starting to visit the lattice of all the rules having a high lift value.

```
std::unordered_set<Rule> RulesFromFrequentItemset::generate_hypotheses(const Pattern<std::string> &
     ↪ pattern) {
    Rule r{pattern};
    return GeneralToSpecificHeuristic::generate_hypotheses(r, scorer.lift(r));
}
```

The execution of this method over each possible frequent itemset remarked in Figure 4 returns the rules $\{E, K\} \Rightarrow \{O\}$, $\{K, O\} \Rightarrow \{E\}$, $\{E\} \Rightarrow \{O\}$, and $\{O\} \Rightarrow \{E\}$.

2.3 Computing the final Result

After implementing both data mining algorithms, we now need to bridge them and to code the CSV parser that is allowing us to read the `cr_data_1535999786739.csv` coming from the `https://github.com/kekepins/clash-royal-analytics/` project. Each row in the CSV file contains the pieces of information pertaining from the first and the second player, and who was the winner between the two.

```
#include <string>
#include <dm/dataset/ClashRoyalePlayer.h>
#include <optional>
#include <limits>  // for csv.h and numeric_limtis
#include <csv.h>

/**
 * Match information concerning who was the winning player.
 */
struct ClashRoyaleDatasetMatch {
    std::string  type;  ///<@ Game match type
    unsigned char winner; ///<@
    std::string  utctime;///<@ Time when the match occured

    ClashRoyalePlayer     player1; ///<@ First player
    ClashRoyalePlayer     player2; ///<@ Second player

    // The [[nodiscard]] attribute can be used to indicate that the return value of a function shouldn
         ↪ 't be ignored when you do a function call. If the return value is ignored, the compiler
         ↪ should give a warning on this.
    [[nodiscard]] const ClashRoyalePlayer& getWinner() const { return winner ? player2 : player1; }
    [[nodiscard]] const ClashRoyalePlayer& getLoser() const { return winner ? player1 : player2; }

    /**
     * Parsing the current line from the CSV file
     * @param csvProjection  CSV file from which the current line is read
     * @return               Structured representation of the CSV line
     */
    static std::optional<ClashRoyaleDatasetMatch> read(
            io::CSVReader<31, io::trim_chars<'␣', '\t'>, io::no_quote_escape<'^'>>& csvProjection);

    // Printing the structured CSV row to an output stream
    friend std::ostream &operator<<(std::ostream &os, const ClashRoyaleDatasetMatch &match);
};
```

Each player has a name, a tag, and belongs to a clan. Any player could have a given amount of trophies that he might lose after the match, and it might also earn crown points. Last, each player is also associated to a deck composed of 8 cards:

```
/**
 * Describing a player participating in a match
 */
struct ClashRoyalePlayer {
    std::string name;           ///< Player's name
    std::string tag;            ///< Player's tag
    std::string clan;           ///< Player's clan
    size_t     startTrophies;   ///< Player's trophies before the match
    size_t     crownsEarned;    ///< How many crowns were earned by the player
    DeckItem   cards[8];        ///< Player's deck at the moment of the battle

    friend std::ostream &operator<<(std::ostream &os, const ClashRoyalePlayer &player);

    /**
     * Represents the deck as a transaction that can be used for the FPGrowth algorithm
     * @return
     */
    Transaction<std::string> asFPTransaction() const {
        Transaction<std::string> toReturn;
        for (size_t i = 0; i<8; i++) toReturn.emplace_back(cards[i].name);
        return toReturn;
    }
};
```

Each card is represented as a `DeckItem` which associates each card name to a strength score. Each DeckItem could be then sorted by strength via relational operators.

```
/**
 * Describing a card within a player's deck
 */
struct DeckItem {
    std::string name;     ///< Card name
    double strength;      ///< Strength associated to the card

    DeckItem() = default;
    DeckItem(const DeckItem& x) = default;
    DeckItem& operator=(DeckItem& x) = default;

    bool operator<(const DeckItem &rhs) const;
    bool operator>(const DeckItem &rhs) const;
    bool operator<=(const DeckItem &rhs) const;
    bool operator>=(const DeckItem &rhs) const;
    friend std::ostream &operator<<(std::ostream &os, const DeckItem &item);
};
```

At this point, we can implement the `read` method from the `ClashRoyaleDatasetMatch` struct similarly to what we have already done for the *Machine Learning* tutorial: each line in the CSV file consists of 31 fields. The deck is represented by a space separated field, which can be then splitted into different card names using a `istringstream` (Lines 19 and 23). Similarly to the previous tutorial, we're going to return the read card (Line 27) only if the parser was able to successfully read the next line (Line 13); otherwise, no card is going to be returned (Line 30).

```
 1 std::optional<ClashRoyaleDatasetMatch>
 2 ClashRoyaleDatasetMatch::read(io::CSVReader<31, io::trim_chars<’␣’, ’\t’>, io::no_quote_escape<’^’>> &
        ↪ csvProjection) {
 3     ClashRoyaleDatasetMatch x;
 4     std::string human_deck_1; ///< This is the deck information: this will be later on splitted if the
            ↪  reading was fine
 5     std::string human_deck_2; ///< Same thing here
 6 
 7     // Reading some general match information
 8     bool test = csvProjection.read_row(x.type, x.winner, x.utctime,
 9             // Information regarding the first player
10             x.player1.name, x.player1.tag, x.player1.clan, x.player1.startTrophies, x.player1.
                   ↪ crownsEarned, human_deck_1, x.player1.cards[0].strength, x.player1.cards[1].
                   ↪ strength, x.player1.cards[2].strength, x.player1.cards[3].strength, x.player1.cards
                   ↪ [4].strength, x.player1.cards[5].strength, x.player1.cards[6].strength, x.player1.
                   ↪ cards[7].strength,
11             // Information regarding the second player
12             x.player2.name, x.player2.tag, x.player2.clan, x.player2.startTrophies, x.player2.
                   ↪ crownsEarned, human_deck_2, x.player2.cards[0].strength, x.player2.cards[2].
                   ↪ strength, x.player2.cards[2].strength, x.player2.cards[3].strength, x.player2.cards
                   ↪ [4].strength, x.player2.cards[5].strength, x.player2.cards[6].strength, x.player2.
                   ↪ cards[7].strength
13             );
14 
15     // Split the deck information in different cards only if the reading was successful
16     if (test) {
17         // Splitting the deck information in a more structured way
18         {
19             std::istringstream s1{human_deck_1};
20             for (auto & card : x.player1.cards) s1 >> card.name;
21         }
22         {
23             std::istringstream s2{human_deck_2};
24             for (auto & card : x.player2.cards) s2 >> card.name;
25         }
26         // Now, we are ready to return the player
27         return {x};
28     } else {
29         // No player was read
30         return {};
31     }
32 }
```

Finally, we can now bridge the two algorithms. As per previous discussion, we want to learn only the successful rules, those are the deck configurations that never lost each recorded battle. In order to do so, we need to distinguish the decks that won at least one match (Line 58) from the others that lost at least one match (Line 59). Then, we obtain the always-winning decks by removing from the set of the winning decks the ones that lost at least one match (Line 77). Given that both the creation of a FPTree (Line 89) and the execution of the FPGrowth algorithm over such tree (Line 92) are going to take some time, we can store the intermediate results of such algorithm in a file named `patterns_file`, so to skip these intermediate steps. Each file's row will separate the support number from the space-separated frequent itemset (Line 100) using a colon (Line 99). Last, we generate and print all the rules generated from each possible frequent itemset having a lift strictly than zero.

```
 1 #include <vector>
 2 #include <csv.h>
 3 #include <ostream>
 4 #include <iostream>
 5 #include <dm/dataset/ClashRoyaleDatasetMatch.h>
 6 #include <fstream>
 7 #include <algorithm>
 8 #include <string>
 9 #include <sstream>
10 
11 #include <set>
12 #include <cmath>
13 #include <dm/DataMiningMetric.h>
14 #include <dm/RulesFromFrequentItemset.h>
15 
16 
17 #ifdef _WIN64
18 #ifndef F_OK
19 #define F_OK    (00)
20 #endif
21 #include <iterator>             //inserter
22 #else
23 extern "C" {
24 #include <unistd.h>
25 }
26 #endif
27 
28 
29 int main(void) {
30     // Loading the Clash Royale dataset. Even though we have more than 200 dimensions, we are only
         ↪ interested in a few of them
31     // This file doesn't fit as it is in GitHub. So, pick the cr_data_1535999786739.zip file from the
         ↪ dataset_clashroyale
32     // submodule, and unzip it in 'data/cr_data_1535999786739.csv'
33     std::string path = "data/cr_data_1535999786739.csv";
34     // File where we can store the previously computed frequent patterns by the FPGrowth algorithm
35     std::string patterns_file = "data/mined_patterns.txt";
36 
37     std::set<Pattern<std::string>> patterns;
38     // Checking if the file exists: if the file doesn't exist, then I need to recompute the frequent
         ↪ itemsets from the database
39     if (!(access(patterns_file.c_str(), F_OK) != -1)) {
40         // Setting ^ as an escape characeter, and using all the possible spaces to be trimmed.
41         io::CSVReader<31, io::trim_chars<'␣', '\t'>, io::no_quote_escape<'^'>> clash_royale{path};
42         // Defining which are the relevant columns to be read, and ignoring the others
43         clash_royale.read_header(io::ignore_extra_column, "type", "winner", "utctime", "name_P1", "
             ↪ tag_P1", "clan_P1", "startTrophies_P1", "crownsEarned_P1", "human_deck_P1", "
             ↪ strenght_P1_0", "strenght_P1_1", "strenght_P1_2", "strenght_P1_3", "strenght_P1_4", "
             ↪ strenght_P1_5", "strenght_P1_6", "strenght_P1_7", "name_P2", "tag_P2", "clan_P2", "
             ↪ startTrophies_P2", "crownsEarned_P2", "human_deck_P2", "strenght_P2_0", "strenght_P2_1"
             ↪ , "strenght_P2_2", "strenght_P2_3", "strenght_P2_4", "strenght_P2_5", "strenght_P2_6",
             ↪ "strenght_P2_7");
44         // Vector storing all the transactions read from the CSV file
45         std::vector<Transaction<std::string>> transactions;
46         {
47             // Shorthand for a set of transaction sorted in lexicographical order
48             using DeckSet = std::set<Transaction<std::string>, VTLexic>;
49             // Separating the decks that wins from the decks that lose some tournaments
50             DeckSet winning_deck, losing_deck;
51             // Reading all the elements in the CSV file
52             std::cout << "Reading␣and␣projecting␣all␣the␣battles␣from␣the␣dataset..." << std::endl;
53             bool continueReading = true;
54             do {
55                 // Read a line from the file if possible
56                 const auto x = ClashRoyaleDatasetMatch::read(clash_royale);
57                 if (x) {                                        // If it was possible to read the
                     ↪ information from the file
```

```
                    winning_deck.emplace(x->getWinner().asFPTransaction());
                    losing_deck.emplace(x->getLoser().asFPTransaction());
                    // Load both the winning and the losing decks
                } else {
                    continueReading = false;                            // Otherwise, fail and stop reading
                }
            } while (continueReading);

            /*
             * Now, we want to obtain the set of decks that are always winning: that information will be then fed to the FPGrowth algorithm
             */
            std::cout << "Initial␣number␣of␣winning␣moves:␣" << winning_deck.size() << std::endl;
            std::cout << "Number␣of␣losing␣decks:␣" << losing_deck.size() << std::endl;
            {
                DeckSet c;
                std::set_difference(std::make_move_iterator(winning_deck.begin()),
                                    std::make_move_iterator(winning_deck.end()),
                                    losing_deck.begin(), losing_deck.end(),
                                    std::inserter(c, c.begin()));
                winning_deck.swap(c);
            }
            std::cout << "Resulting␣number␣of␣always␣winning␣decks:␣" << winning_deck.size() << std::endl;
            std::copy(winning_deck.begin(), winning_deck.end(), std::back_inserter(transactions));
        }
        std::cout << std::endl << std::endl;

        size_t minimum_support_threshold = 10000; // The pattern is frequent, and therefore supported by at least 10,000 rounds
        std::cout << "Feeding␣the␣FPGrowth␣Algorithm..." << std::endl;

        {
            std::cout << "a)␣creating␣the␣tree" << std::endl;
            const FPTree<std::string> fptree{ transactions, minimum_support_threshold };

            std::cout << "b)␣computing␣the␣most␣frequent␣patterns" << std::endl;
            patterns = fptree_growth( fptree );
        }

        {
            std::cout << "c)␣Saving␣all␣the␣patterns␣in␣the␣file␣as␣a␣backup" << std::endl;
            std::ofstream mined_patterns{patterns_file};
            for (const Pattern<std::string>& pattern : patterns) {
                mined_patterns << pattern.second << "␣:␣";
                for (const std::string& item : pattern.first) {
                    mined_patterns << item << "␣";
                }
                mined_patterns << std::endl;
            }
        }
    } else {
        // ... If the file exists, then:
        std::cout << "Patterns␣have␣been␣already␣mined␣in␣a␣previous␣computation:␣loading␣those␣from␣the␣file!" << std::endl;
        // Opening the existing file containing the patterns
        std::ifstream mined_patterns{patterns_file};

        // The variable containing each line from the file
        std::string line;

        // Reading all the lines
        while (std::getline(mined_patterns, line)) {
            std::istringstream iss(line);
            uint64_t frequency;
            std::string item;
            char delimiter;

            // The first element in the row is the frequency value and some delimiter.
            iss >> frequency >> delimiter;
            // Assertion on the delimiter separating the frequency from the items
            assert(delimiter = ':');

            // Storing all the elements in the pattern
            std::set<std::string> pattern;
            while (iss >> item) pattern.emplace(item);

            // Storing the resulting pair in the set
            Pattern<std::string> mined{pattern, frequency};
            patterns.emplace(mined);
        }
    }

    /*
     * Initializing the scoring functions using the mined patterns from the FPGrowth algorithm
     */
    DataMiningMetrics counter{patterns};
    size_t count = 0;

    // For each frequent itemset, generate a set of possible relevant rules
    for (const Pattern<std::string>& pattern : patterns) {
        if (pattern.first.size() <= 1) continue;                            // I cannot extract any
             ↪ significant rule from a singleton!

        RulesFromFrequentItemset rffi{counter};                         // Generate the top rule
        for (auto& result: rffi.generate_hypotheses(pattern)) {            // Generate the hypotheses
             ↪ containing a lift greater than one
            std::cout << result << std::endl;                              // Printing each successful
                 ↪ outcome
            count++;
        }
    }

    std::cout << std::endl << "~␣We␣mined␣" << patterns.size() << "␣patterns␣" << std::endl;
    std::cout << "~␣from␣which␣we␣generated␣" << count << "␣rules!" << std::endl;
}
```

3 Further Work

1. How does calculating $\mathbf{F_i}$ from $\mathbf{F_{i-1}} \times \mathbf{F_{i-1}}$ helps optimizing the naive Apriori approach? Look up [4] for a hint.

2. With respect to the FPTree and FPGrowth implementation:

   (a) Change the code so that instead of double scanning each transaction alongside with I sorted by decreasing support, we directly sort each transaction. Does this change improve or reduce the efficiency?

   (b) Replace the `count` method invocation at Line 101 on page 9 with an `insert` (**Hint**: such method returns a pair, where the second value is a boolean determining if the iterator returned by the first value of the pair is a pointer to a newly inserted object or not).

   (c) Optimize the implementation of the `header_table` so you are not forced to scan the whole `node_link` as in Line 103 on page 9 to add another element (**Hint**: use an end pointer!).

   (d) Parallelize the `header_table` scan in Line 52 on page 11 so that each single thread will be associated to one single frequent item $i \in I$: nevertheless, this should be done only when $\varsigma = \varepsilon$.

References

[1] Flavio Bertini, Giacomo Bergami, Danilo Montesi, Giacomo Veronese, Giulio Marchesini, and Paolo Pandolfi. Predicting frailty condition in elderly using multidimensional socioclinical databases. *Proceedings of the IEEE*, 106(4):723–737, 2018. `doi:10.1109/JPROC.2018.2791463`.

[2] Andre Petermann, Giovanni Micale, Giacomo Bergami, Alfredo Pulvirenti, and Erhard Rahm. Mining and ranking of generalized multi-dimensional frequent subgraphs. In *Twelfth International Conference on Digital Information Management, ICDIM 2017, Fukuoka, Japan, September 12-14, 2017*, pages 236–245, 2017. `doi:10.1109/ICDIM.2017.8244685`.

[3] Luc De Raedt. *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies).* Springer-Verlag, Berlin, Heidelberg, 2008.

[4] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2005.

A Formal Proofs

Lemma 1. *Given two rules $X \to Y$ and $T \to U$, we have that $T \to U \preceq X \to Y$ if and only if $T \subseteq X$ and $U \supseteq Y$.*

Proof. From the definition provided in the second section, we provided the definition of the most general unifier. Given that $T \to U$ is also the most general unifier between this rule and $X \to Y$ ($mgg(T \to U, X \to Y) = T \to U$), then from the definition of such general unifier we can observe that $X \cap T = T$ and that $Y \cup U = U$. We observed in the former tutorial that the intersection with a subset is a subset, so we have that $T \subseteq X$. After observing that the union with a superset is a superset[7], we can also determine that $Y \subseteq U$. □

Lemma 2. *The min lift quality predicate is anti-monotonic w.r.t. an association rule $X \to Y$.*

Proof. In Lemma 2 from the *Modelling Learning* tutorial, we proved that *min-support* quality criterion is an anti-monotonic with respect to a given data set $D \subseteq \mathcal{L}_e$. Now, we will show that the minimum lift measure is also anti-monotonic within the lattice created from one seed rule from each frequent itemset. In particular, after recalling Lemma 1 from the same tutorial, we can rewrite that:

$$Z \to T \preceq X \to Y \Leftrightarrow X \subseteq Z \wedge T \subseteq Y$$

After observing that $X \cup Y = Z \cup T$ for construction and then $\sigma(X \cup Y) = \sigma(Z \cup T) = c$, we can establish the relationship between *lift*$(X \to Y)$ and *lift*$(Z \to Y)$ as follows:

$$\mathit{lift}(Z \to T) = \frac{\sigma(Z \cup T)}{\sigma(Z)\sigma(T)} = \frac{c}{\sigma(\mathbf{Z})\sigma(T)} < \frac{c}{\sigma(\mathbf{X})\sigma(\mathbf{T})} < \frac{c}{\sigma(X)\sigma(\mathbf{Y})} = \mathit{lift}(X \to Y)$$

$$\mathit{lift}(Z \to T) < \mathit{lift}(X \to Y)$$

from which it descends that if *lift*$(Z \to T) > \theta$, then also *lift*$(X \to Y) > \theta$. □

B Lexicographical Order

```
#define ForAll2(T,U)    template <typename T, typename U>

/**
 * Defining an ordering over vectors following the lexicographical order
 * @tparam T   Type associated to the container having a operator[ and a size method
 * @tparam U   Type associated to the content
 */
ForAll2(T,U) struct LexicographicalOrder {
    std::less<U> lesscmp; // Default comparator for the content
    bool operator()(const T& lhs, const T& rhs) const {
        return compare(lhs, rhs, 0);
    }
private:
    bool compare(const T& lhs, const T& rhs, size_t idx) const {
        if (idx == lhs.size()) {
            return !(idx == rhs.size());
        } else {
            if (idx == rhs.size()) {
                return false;
            } else {
                if (lesscmp(lhs[idx], rhs[idx])) {
                    return true;
```

[7] https://proofwiki.org/wiki/Union_with_Superset_is_Superset.

```
                } else if (lesscmp(rhs[idx], lhs[idx])) {
                    return false;
                } else {
                    return compare(lhs, rhs, idx+1);
                }
            }
        }
    }
};
```