# Project Plan Description

## Introduction

### Abstract

In class, we were introduced to the problem of frequent itemset mining. In simplest form, the goal of frequent itemset mining is to discover *sets of items* (itemsets) *that appear frequently* (defined by support). One of the biggest limitations of frequent itemset mining is that it assumes that all itemsets are equally valuable. For instance, suppose that in a dataset of grocery items, we find that the itemset $\{milk, bread\}$ is very frequent. This transaction, while frequent, does not really have any value to us since these are both lower cost items in comparison to, for instance, $\{champagne, caviar\}$.

This limitation called for a generalization of the frequent itemsets problem to **high utility itemset mining.** The fundamental difference between frequent itemset mining and high utility itemset mining is that when we are performing high utility itemset mining, every item has an associated weight describing how profitable the item is. High utility itemset mining also considers the quantity of the items in the database.

In this project, we plan to provide a preliminary introduction to the field of high utility itemset mining by implementing and evaluating the performance of two separate high utility itemset mining algorithms and evaluate their performance

### Definitions

Before continuing with our algorithms, we define terminology that is essential to understand our algorithms and their performance.

1. **Transaction Database:** A set of records (transactions) indicating the items purchased by customers at different times. Typically, transactions are represented as rows in a dataset.
2. **Quantitative Transaction Database:** A transaction database that includes the quantities of items in transactions and weights indicating the relative importance of each item to the user.
3. **Support measure:** The support of an itemset $X$ in a transaction database $D$ is defined as $sup(X) = |\{T \mid X \subseteq T \wedge T \in D)|$, or the number of transactions that contain $X$.
4. **Frequent itemset:** An itemset is frequent if its support $sup(X)$ is no less than the *minsup* threshold.
5. **Utility:** The utility of an item $i$ in a transaction $T_c$ is denoted as $u(i, T_c)$ and is defined as $u(i, T_c) = p(i) \times q(i, T_c)$, where $p(i)$ denotes the external utility, or weight, of the item, and $q(i, T_c)$ denotes the internal utility, or the quantity of $i$ in the transaction $T_c$.
6. **High-utility itemset:** An itemset $X$ is a *high-utility itemset* if its utility $u(X)$ is no less than a user-specified minimum utility threshold *minutil* set

by the user.

7. **The TWU measure:** The *transaction utility* ($TU$) of a transaction $T_c$ is the sum of the utilities of all the items in $T_c$. The *transaction-weighted utilization* ($TWU$) of an itemset $X$ is defined as the sum of the transaction utilities of transactions containing $X$.

## Algorithms

### Two-Phase

Test

### FHM

FHM is a one-phase algorithm for high-utility itemset mining. The main algorithm takes a quantitative transaction database and the *minutil* threshold as input. Then, FHM scans the database to calculate the TWU of each item and creates the set $I^*$, which contains all items having a TWU no less than *minutil*. We define a total order $\succ$ as the order of ascending TWU values. Another database scan is performed, where items in transactions are reordered according to $\succ$, the utility-list of each item in $I^*$ is built, and a structure named EUCS (Estimated Utility Co-occurrence Structure) is created.

A utility-list for an itemset is a set of tuples for each transaction. Each tuple is of the form $(tid, iutil, rutil)$, where *tid* is the transaction ID, *iutil* is the utility of the itemset in the transaction, and *rutil* is the total utility of all the items in the transaction that have a TWU greater than those in the itemset. As we will seen, utility-lists allow us to quickly calculate the utility of an itemset and upper-bounds on the utility of its supersets. Additionally, utility-lists for itemsets with size greater than 1 can be quickly created by joining utility-lists of smaller itemsets. The EUCS is defined as a set of triples of the form $(a, b, c) \in I^* \times I^* \times \mathbb{R}$ such that $\text{TWU}(\{a, b\}) = c$. The EUCS is the main novelty in FHM which allows for the pruning mechanism named EUCP (Estimated Utility Co-occurrence pruning) that will be mentioned later.

After the EUCS is created, a recursive depth-first search of the itemsets is performed. The search algorithm, called *FHMSearch*, takes as input (1) an itemset $P$, (2) a set of extensions of $P$ with the form $Pz$ where $Pz$ was created by appending item $z$ to $P$, (3) *minutil*, and (4) the EUCS. The first call to *FHMSearch* gives an empty set for the itemset, $I^*$ for the set of extensions of the itemset, *minutil*, and the EUCS. *FHMSearch* executes as follows. For each extension $Px$ of $P$, if the sum of the *iutil* values for $Px$'s utility-list (which is equal to the utility of $Px$) is no less than *minutil*, then $Px$ is a high-utility itemset and it is output. We do not consider extensions of $P$ that have utilities less than *minutil* because these extensions are by definition low-utility itemsets. Next, if the sum of *iutil* and *rutil* values in $Px$'s utility-list are no less than *minutil*, then the extensions of $Px$ are explored. We do not explore extensions

of $Px$ if the sum of *iutil* and *rutil* values in $Px$'s utility-list is less than *minutil* because the extensions of $Px$ and their supersets are low-utility itemsets. We then explore the extensions of $Px$ by considering all extensions $Py$ of $P$ such that $y \succ x$. If there exists $(x, y, c)$ in the EUCS (i.e., TWU$(\{x, y\}) = c$) such that $c \geq minutil$, then we merge $Px$ with $Py$ to generate extensions of the form $Pxy = P \cup \{x, y\}$. We do not generate $Pxy$ if $c$ is less than *minutil* because this would mean that $Pxy$ and all its supersets are low-utility itemsets. This step is key to EUCP as it avoids the costly join operation to calculate the utility-list of an itemset that is detailed below.

To construct the utility-list of $Pxy$ the *Construct* algorithm is called to join the utility-lists of $P$, $Px$, and $Py$. *Construct* takes $P$, $Px$, and $Py$ as input an executes as follows. The utility-list of $Pxy$ is initialized as an empty set. Next, for each tuple $ex$ in the utility-list of $Px$, if there exists a tuple $ey$ in the utility-list of $Py$ such that $ex.tid = ey.tid$, and the utility-list of $P$ is empty, then $exy$ – the tuple for $Pxy$'s utility-list – is formed as $(ex.tid, ex.iutil + ey.iutil, ey.rutil)$. Note that $P$ being empty implies that $Px = \{x\}$ and $Py = \{y\}$. If the utility-list of $P$ is non-empty, then we search for the tuple $e$ in the utility-list of $P$ such that $e.tid = ex.tid$ and create $exy$ as $(ex.tid, ex.iutil + ey.iutil - e.iutil, ey.rutil)$. After $exy$ is created, it is appended to the utility-list for $Pxy$. Once we have considered all the tuples in the utility-list of $Px$, we return the utility-list of $Pxy$, which terminates the execution of *Construct*.

After we have created the utility-list of $Pxy$, a recursive call to *FHMSearch* with $Pxy$ is done to calculate its utility and explore its extension(s). Starting with single items, *FHMSearch* recursively explores the search space of itemsets by appending single items until all high-utility itemsets are discovered.

## Experiments

## Work

// TODO: split the following items into smaller steps

- Write code for the Two-Phase algorithm.
- Write code for the FHM algorithm.
- Write experiment scripts to evaluate both algorithms.
- Write content for the final video.
- Produce the final video.

## Plan

// TODO: describe how we will submit the following

Submit by phase 3:

- Code for the Two-Phase algorithm.
- Code for the FHM algorithm.

Submit by phase 4:

- Code for the Two-Phase algorithm.
- Code for the FHM algorithm.
- Code for the experiment scripts
- Results of experiments
- Final video

## Logistics

// TODO: decide who will do what

We have a GitHub repository to share code and data among the members of the group.

We plan to split up the work as follows. The first member's main responsibility is to write the code for the Two-Phase algorithm. Similarly, the second member's main responsibility is to write the code for the FHM algorithm. The third member's main responsibility is to write the experiments that evaluate the two algorithms. The fourth member's main responsibility is to edit/produce the final video as well as help with smaller tasks that the first three members may have. All members are also responsible for writing content for the final video.

The timeline for our project is described below:

- April 19 (end of phase 3): finish writing the Two-Phase and FHM algorithms.
- April 30: finish writing experiment scripts and evaluating both algorithms.
- May 7: finish writing content for the final video.
- May 21: finish making the final video.