Final Report

Michael Bakshandeh

5/19/2021

Background

This report details our findings from running a variety of datasets on both the Two-Phase and FHM high utility itemset mining (HUIM) algorithms. We encountered runtime issues when trying to test Two-Phase on larger datasets, which made sense since our Two-Phase implementation relied on completely reading and recomputing TWUs as we generated more candidates. As a result, most of our comparisons between the two algorithms will be based on the `foodmart` dataset, which was a real, small-scale database. We will also discuss FHMs performance across different dataset sizes.

Computing Appropriate Threshold Values

One of the challenges of conducting these experiments was computing appropriate *minutil* values. Algorithm-specific papers used the **minimum utility threshold** to determine appropriate threshold values.

Formally, we define **minimum utility threshold** δ as the percentage of total transaction utility values in the database, i.e., $minutil = \delta * \sum_{T_q \in D} tu(T_q)$, where T_q represents the q^{th} transaction in quantitative transaction database D and $tu(T)$ represents the utility of transaction T.

In literature, the way that experiments are run is that δ is fixed at 1 and is consistently lowered until the algorithm takes too long to compute rationally. To simplify the amount of work we had to do, we based our threshold values θ directly from the literature rather than experimentally lowering δ until failure. The x-axis of all graphs in the report represents the actual *minutil* values as a result of the computation listed in the definition above, rather than the δ values themselves.

Comparison between FHM and Two-Phase

As detailed in our Project Proposal, we implemented two different HUIM algorithms, Two-Phase and FHM. Two-Phase is the equivalent of an apriori algorithm, while FHM is meant to be a significant improvement on FHM by reducing runtime and memory usage. To formalize these theoretical results, we ran each algorithm on the `foodmart` dataset.

Description of Dataset

The `foodmart` dataset is a dataset of customer transactions from a retail store, obtained and transformed from SQL-Server 2000. The dataset contains 4,141 transactions, 1,559 items, and an average of 4.42 items per transaction.

Graphs

The survey paper along with specific paper algorithms used several different benchmarks for measuring the performance of the algorithms, namely runtime, memory usage, and pruned itemset count. We also looked into other metrics which will be discussed after.

Dataset Creation

```
experiments_foodmart_fhm <- read.csv("experiments/experiment_food_mart.csv")
head(experiments_foodmart_fhm)
```

```
##   minimum.utility total.runtime..ms. high.utility.itemSet.count
## 1           12011           555.0473                        258
## 2            9609           580.2171                        468
## 3            7206           642.5450                       1487
##   candidate.itemSet.count pruned.itemSet.count maximum.memory.used..MB.
## 1                     129                38421                 23.21875
## 2                     539                38582                 23.22266
## 3                    6705                41026                 23.20312
```

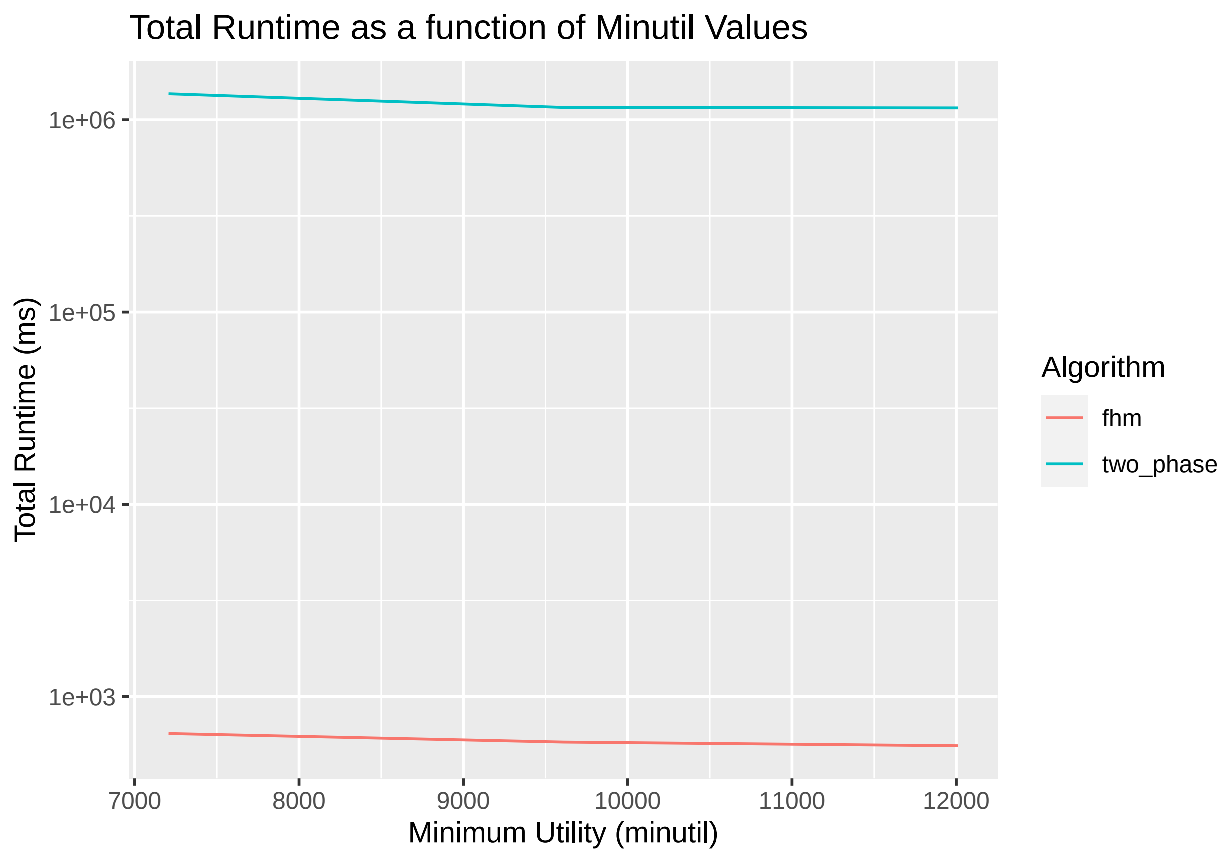
```
experiments_foodmart_fhm$algo <- "fhm"
experiments_foodmart2 <- read.csv("experiments/experiment_food_mart2.csv")
experiments_foodmart2$algo <- "two_phase"
foodmart <- rbind(experiments_foodmart_fhm, experiments_foodmart2)
head(foodmart) # to show what our database looks like
```

```
##   minimum.utility total.runtime..ms. high.utility.itemSet.count
## 1           12011           555.0473                        258
## 2            9609           580.2171                        468
## 3            7206           642.5450                       1487
## 4           12011       1152918.4420                        258
## 5            9609       1159555.3112                        468
## 6            7206       1365880.0960                       1487
##   candidate.itemSet.count pruned.itemSet.count maximum.memory.used..MB.
## 1                     129                38421                 23.21875
## 2                     539                38582                 23.22266
## 3                    6705                41026                 23.20312
## 4                    1729                38346                155.76172
## 5                    2216                38141                157.03906
## 6                   43547                36896                157.16016
##        algo
## 1       fhm
## 2       fhm
## 3       fhm
## 4 two_phase
## 5 two_phase
## 6 two_phase
```

Runtime Comparison

The graph below shows the difference between runtimes for Two-Phase and FHM.

```
# runtime
ggplot(foodmart, aes(x=minimum.utility, y=total.runtime..ms.)) +
  geom_line(aes(color=algo)) +
  scale_y_continuous(trans='log10') +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Total Runtime (ms)",
    title = "Total Runtime as a function of Minutil Values",
    color = "Algorithm"
  )
```

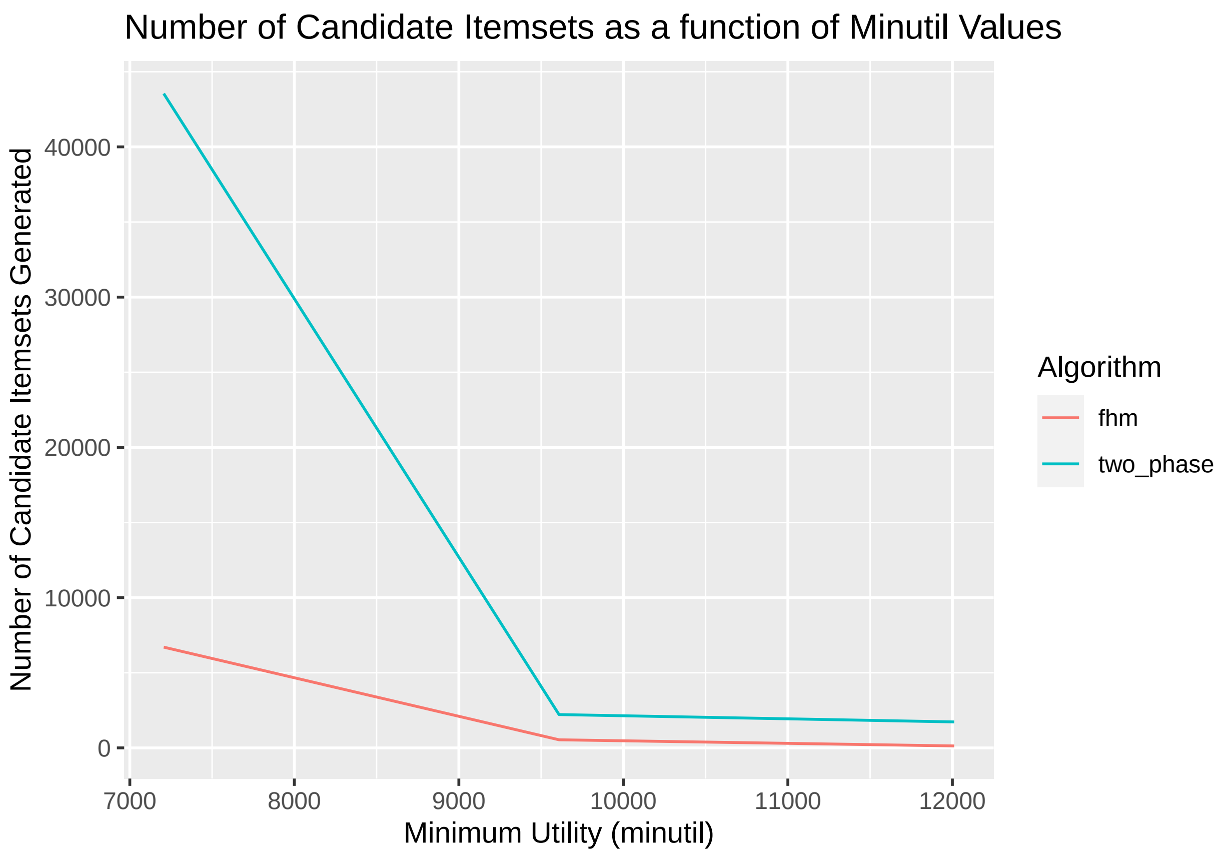


A quick look at the graph tells us that, as theoretical findings would suggest, the total runtime tended to decrease as the *minutil* increased, but the rate of change to the decrease appeared to fall as the minimum utility threshold increased (i.e., *minutil* increased and got closer to the total utility of the database). Furthermore, it appears that FHM was orders of magnitude more efficient than Two-Phase. We can see that Two-Phase never took less than 1,000,000 ms = 1,000 seconds = 16.67 minutes to complete, while FHM never took more than 500 ms = 0.5 seconds to complete.

Candidate Itemsets

Graphing the number of candidate itemsets explains why the runtimes for the two algorithms were vastly different from each other.

```
# number of candidate itemsets
ggplot(foodmart, aes(x=minimum.utility, y=candidate.itemSet.count)) +
  geom_line(aes(color=algo)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Number of Candidate Itemsets Generated",
    title = "Number of Candidate Itemsets as a function of Minutil Values",
    color = "Algorithm"
  )
```

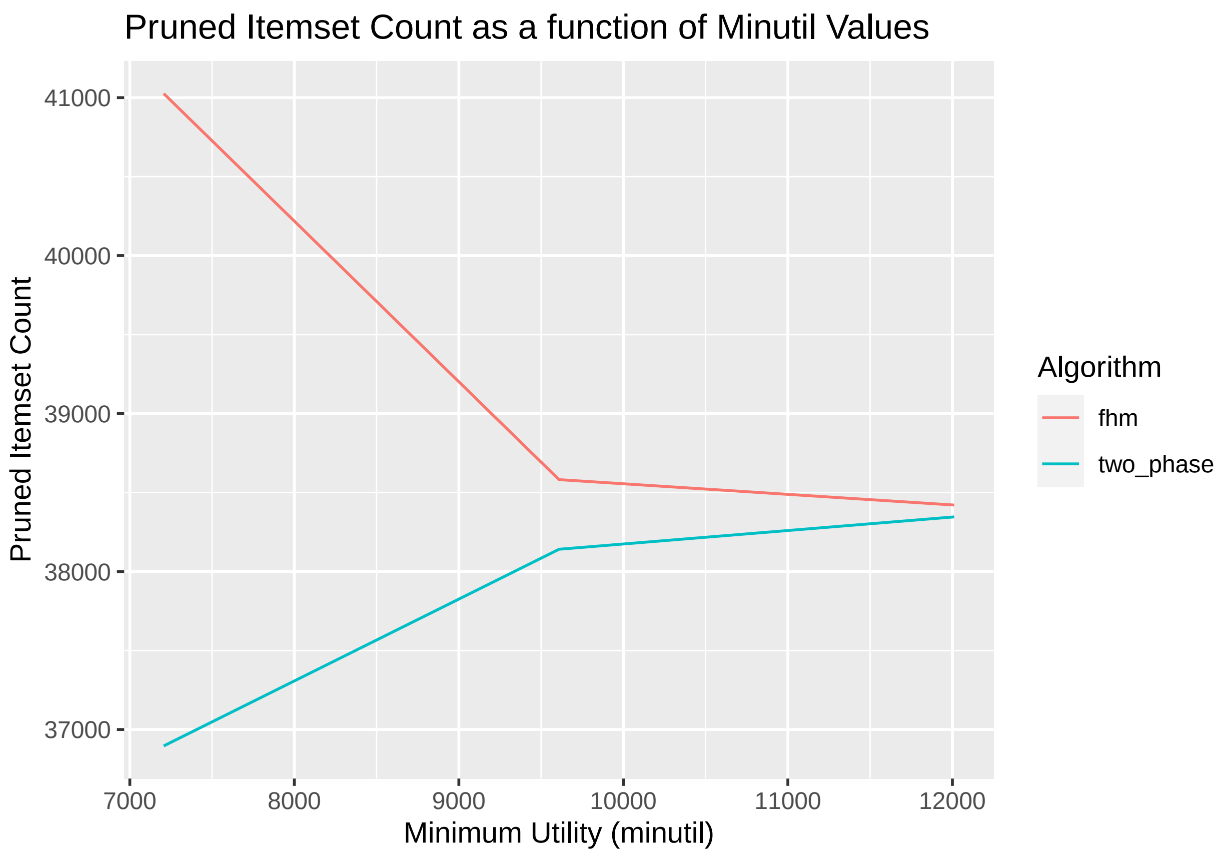


We notice that for smaller *minutil* values, the number of candidates generated is orders of magnitude different while for larger *minutil* values, the number of candidates generated between the two algorithms appears to be very similar. This finding lines up with our theoretical understanding of the two algorithms, since one of the biggest problems of Two-Phase is that it often generates unreasonable candidates. While it is surprising that the number of generated candidates between the two algorithms converged as *minutil* increased, it makes sense that Two-Phase is still more inefficient since it requires many more passes through the dataset than FHM does.

Pruned Itemset Count

The graph below shows the pruned itemset count for each algorithm.

```r
# pruned itemset count
ggplot(foodmart, aes(x=minimum.utility, y=pruned.itemSet.count)) +
  geom_line(aes(color=algo)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Pruned Itemset Count",
    title = "Pruned Itemset Count as a function of Minutil Values",
    color = "Algorithm"
  )
```



Again, we notice that at smaller values of *minutil*, FHM does a significantly better job in terms of efficiency since it successfully prunes ~6,000 more candidates than Two-Phase. Similar to the number of candidates, we notice that the number of pruned itemsets converges as the *minutil* increases.

Memory Usage

To compute memory usage, we used the memory profiler library from Python. The table below shows the memory usage for each algorithm at various sample sizes:

```
memusage <- foodmart %>%
  select(algo, minimum.utility, maximum.memory.used..MB.)
kbl(memusage, booktabs = T, caption = "Memory Usage for FHM and Two Phase") %>%
  kable_styling(latex_options = c("striped", "hold_position"),
                full_width = F)
```

Table 1: Memory Usage for FHM and Two Phase

algo	minimum.utility	maximum.memory.used..MB.
fhm	12011	23.21875
fhm	9609	23.22266
fhm	7206	23.20312
two_phase	12011	155.76172
two_phase	9609	157.03906
two_phase	7206	157.16016

We notice that there is very little within algorithm variation for memory usage, but Two-Phase clearly uses more memory than FHM. This is likely because Two-Phase is a breadth first seach algorithm and thus has to store all candidates it finds in a list. On the other hand, FHM is a depth first search algorithm, which uses a itemset buffer (of fixed memory) to store multiple candidates.

FHM Specific Analysis

We also were able to run FHM on different datasets to show how the algorithm responds to different datasets. We will be specifically looking at the runtime for these results since we did a cross comparison for all important metrics on `foodmart` above.

Datasets

We will be considering 4 different datasets in this section:

1. `chainstore:` customer transactions from a major grocery store chain in California, USA, containing 1,112,949 transactions, 46,086 items, and an average of 7.23 items per transaction, obtained and transformed from NU-Mine Bench.
2. `retail:` customer transactions from an anonymous Belgian retail store, containing 88,162 transactions, 16,470 items, and an average of 10.30 items per transaction.
3. `kosarak:` transactions from click-stream data from an Hungarian news portal, containing 990,002 transactions, 41,270 items, and an average of 8.1 items per transaction.
4. `chess:` prepared based on the UCI chess dataset, containing 3,196 transactions, 75 items, and an average of 37 items per transaction.

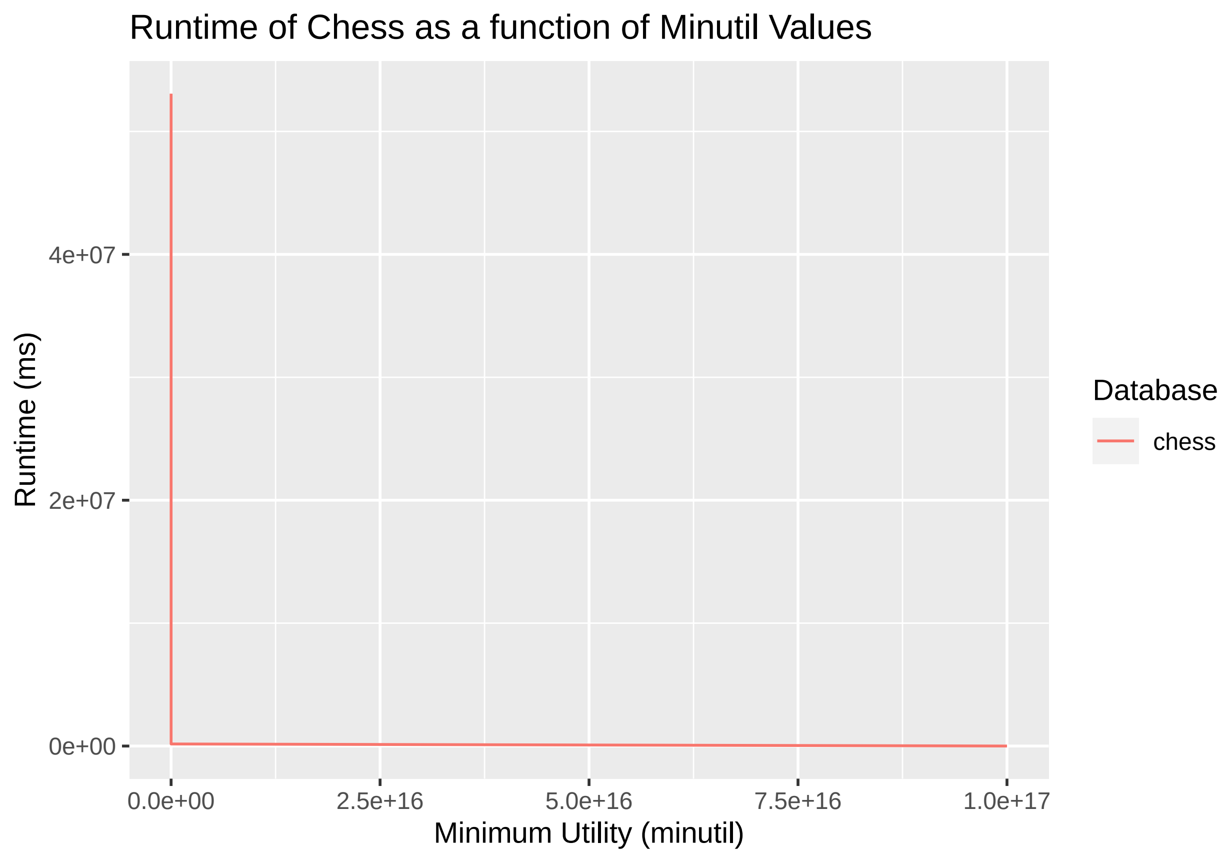
Runtime Graphs

We will be graphing the runtimes for each algorithm below:

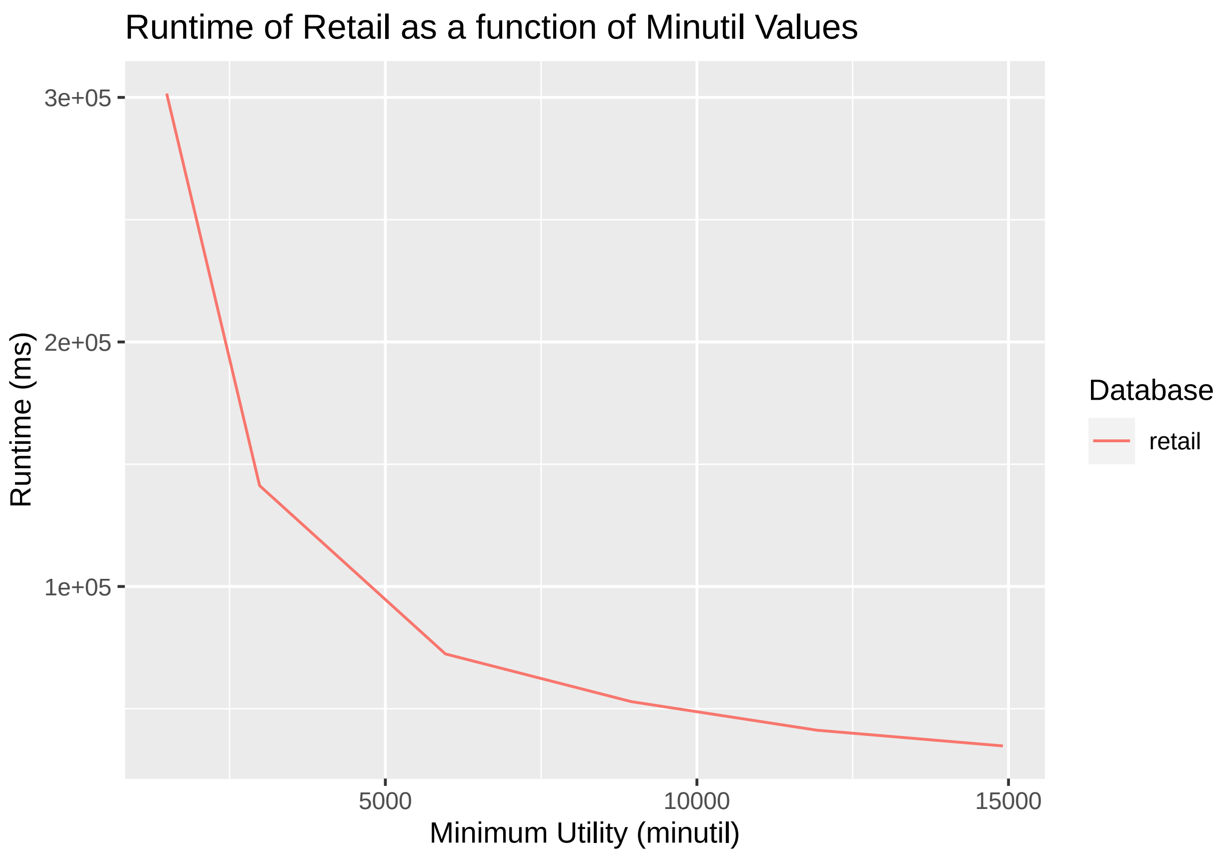
```r
chain_store <- read.csv("experiments/experiment_chain_store.csv")
chain_store1 <- chain_store %>%
  select(minimum.utility, total.runtime..ms.)
chain_store1$db <- "chainstore"

retail <-read.csv("experiments/experiment_retail.csv")
retail1 <- retail %>%
  select(minimum.utility, total.runtime..ms.)
retail1$db <- "retail"

kosarak <- read.csv("experiments/experiment_kosarak.csv")
kosarak1 <- kosarak %>%
  select(minimum.utility, total.runtime..ms.)
kosarak1$db <- "kosarak"

chess <- read.csv("experiments/experiment_chess.csv")
chess1 <- chess %>%
  select(minimum.utility, total.runtime..ms.)
chess1$db <- "chess"

temp <- rbind(chain_store1, retail1)
temp1 <- rbind(temp, kosarak1)
experiments <- rbind(temp1, chess1)
experiments <- experiments %>%
  group_by(minimum.utility, db) %>%
  summarize(runtime = mean(total.runtime..ms.))
```

```
## `summarise()` has grouped output by 'minimum.utility'. You can override using the `.groups` argument
```

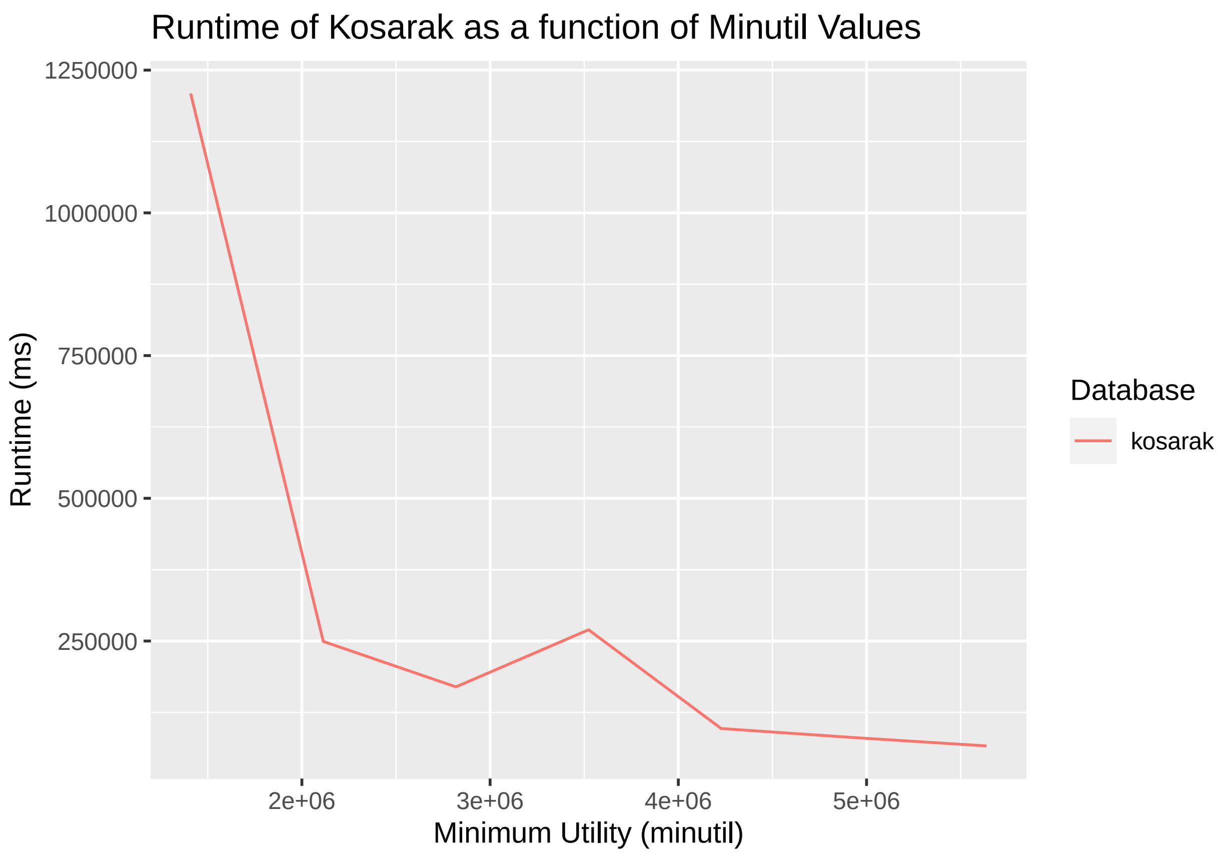
```
# graph the runtime of each algorithm
chess <- experiments %>% filter(db == 'chess')
ggplot(chess, aes(x=minimum.utility, y=runtime)) +
  geom_line(aes(color=db)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Runtime (ms)",
    title = "Runtime of Chess as a function of Minutil Values",
    color = "Database"
  )
```



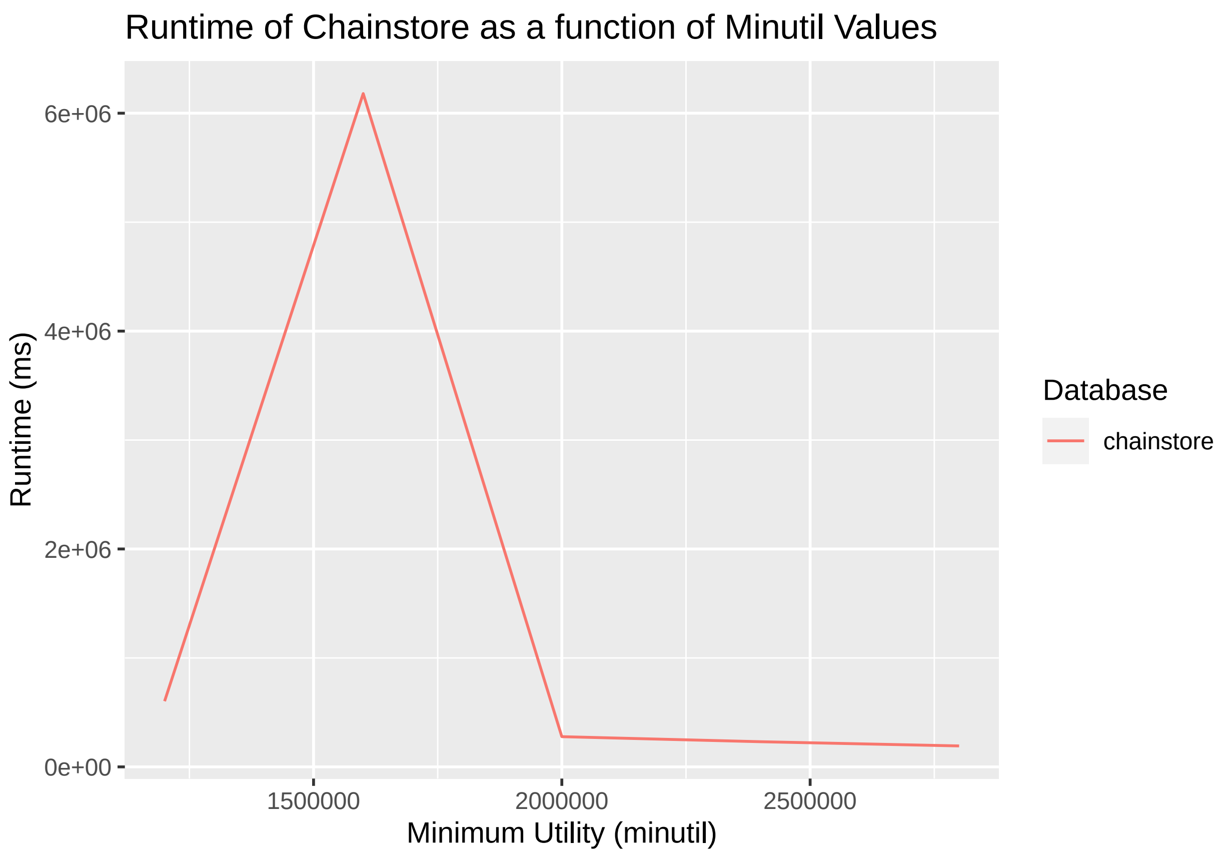
```
retail <- experiments %>% filter(db == 'retail')
ggplot(retail, aes(x=minimum.utility, y=runtime)) +
  geom_line(aes(color=db)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Runtime (ms)",
    title = "Runtime of Retail as a function of Minutil Values",
    color = "Database"
  )
```



```r
kosarak <- experiments %>% filter(db == 'kosarak')
ggplot(kosarak, aes(x=minimum.utility, y=runtime)) +
  geom_line(aes(color=db)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Runtime (ms)",
    title = "Runtime of Kosarak as a function of Minutil Values",
    color = "Database"
  )
```



```r
chain_store <- experiments %>% filter(db == 'chainstore')
ggplot(chain_store, aes(x=minimum.utility, y=runtime)) +
  geom_line(aes(color=db)) +
  labs(
    x = "Minimum Utility (minutil)",
    y = "Runtime (ms)",
    title = "Runtime of Chainstore as a function of Minutil Values",
    color = "Database"
  )
```



The graphs above appear to partially reaffirm what should theoretically happen as we increase the minutil. There are certain graphs that have strange looking graphs (for instance, chess) for reasons that we are not aware of. We did expect chess to have one of the largest runtimes since chess is a dense dataset while the others are sparse, but the drop off in runtime should not be that steep. We also notice that there are some graphs like kosarak in which the runtime unexpectedly rises as the minimum utility increases, which could partially be due to unexpected changes in CPU usage on our personal laptops.