

Report: Assignment #1

ITE4005, Data Science.

2016025305, Jihun Kim<jihunkim@hanyang.ac.kr>.

Finding association rules using **Apriori** algorithm.

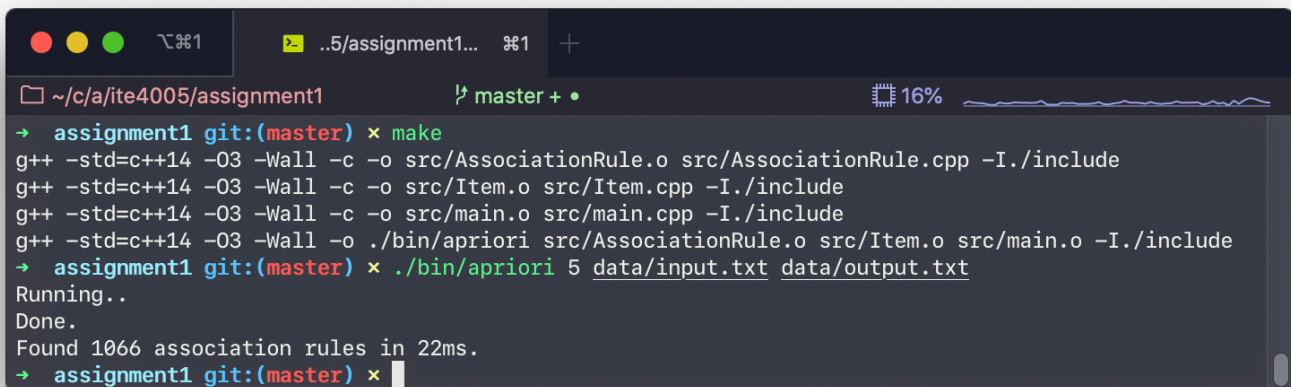
Getting Started

Development Environment

- * **OS:** macOS 10.15.4 (x86_64-apple-darwin19.4.0)
- * **Language:** C++14
- * **Compiler:** Apple clang version 11.0.3 (clang-1103.0.32.29)

Compile & Run

```
$ cd /path/to/repo
$ make
$ ./bin/apriori 5 data/input.txt data/output.txt
$ vi data/output.txt
```



The screenshot shows a terminal window with the following content:

```
~/c/a/ite4005/assignment1  master + • 16%
→ assignment1 git:(master) x make
g++ -std=c++14 -O3 -Wall -c -o src/AssociationRule.o src/AssociationRule.cpp -I./include
g++ -std=c++14 -O3 -Wall -c -o src/Item.o src/Item.cpp -I./include
g++ -std=c++14 -O3 -Wall -c -o src/main.o src/main.cpp -I./include
g++ -std=c++14 -O3 -Wall -o ./bin/apriori src/AssociationRule.o src/Item.o src/main.o -I./include
→ assignment1 git:(master) x ./bin/apriori 5 data/input.txt data/output.txt
Running..
Done.
Found 1066 association rules in 22ms.
→ assignment1 git:(master) x
```

Implementation

My implementation use directed acyclic graph (DAG) for frequent pattern mining.

Each node in DAG represents frequent itemset, and if k -itemset u is a subset of $k + 1$ -itemset v , then u becomes child node of v .

DAG approach reduced tremendous of computation because it continuously holds its child.

ItemSet & Item Representation

```
1 class ItemSet {
2 public:
3     virtual ~ItemSet() = default;
4
5     size_t size() const;
6     size_t num_children() const;
7     size_t support();
8
9     void add_child(ItemSet *itemset);
10    static ItemSet *get_union(ItemSet *l, ItemSet *r);
11    virtual std::set<ItemSet *> get_descendants();
12
13    bool operator==(const ItemSet &o) const;
14    bool operator<(const ItemSet &o) const;
15
16    std::vector<item_id_t> items;
17
18 protected:
19     ItemSet() = default;
20
21     bool txns_updated = false;
22     std::vector<txn_id_t> txns;
23
24 private:
25     std::set<ItemSet *> children;
26     std::set<ItemSet *> descendants;
27 };
```

class ItemSet Represents an itemset. It holds

- vector of transactions (**txns**)
- set of pointer to children (**children**)
- set of pointer to descendants (**descendants**)

And we have another class, **Item**, which is a special case of itemset that has only one item. It has neither children nor descendants.

```
1 class Item : public ItemSet {
2 public:
3     explicit Item(item_id_t item_id);
4     ~Item() override = default;
5
6     void add_transaction(txn_id_t txn_id);
7     std::set<ItemSet *> get_descendants() override;
8 };
```

Compute Union of Two Itemsets

```
1 ItemSet *ItemSet::get_union(ItemSet *l, ItemSet *r) {
2     auto result = new ItemSet;
3     std::set_union(l->items.begin(), l->items.end(),
4                   r->items.begin(), r->items.end(),
5                   std::back_inserter(result->items));
6     result->children.insert(l);
7     result->children.insert(r);
8     return result;
9 }
```

Get union of two itemsets (to generate candidates).

Union items in two itemsets, and add them in children set of newly made parent itemset.

Because it is costly, this function doesn't compute intersection of transactions in two itemsets. This computation will be done lazily. See the code below.

```
1 size_t ItemSet::support() {
2     if (!txns_updated) {
3         std::set_intersection((*children.begin()->txns.begin(), (*children.begin()->txns.end(),
4                           (++children.begin()->txns.begin(), (++children.begin()->txns.end(),
5                           std::back_inserter(txns));
6         txns_updated = true;
7     }
8     return txns.size();
9 }
```

Get All Descendants

```
1 std::set<ItemSet *> ItemSet::get_descendants() {
2     if (descendants.empty()) {
3         for (auto &child : children) {
4             auto child_descendants = child->get_descendants();
5             descendants.insert(child_descendants.begin(), child_descendants.end());
6             descendants.insert(child);
7         }
8     }
9     return descendants;
10 }
```

Main use of this function is to find association rules via descendant traversal.

It recursively call same function of childs to get descendants, and store it to prevent redundant computation.

Note that duplication of descendants from different children can exist, so **std::set** is used.

Reading Items from Input File Stream

```
1 vector<Item *> read_items(ifstream &ifs) {
2     unordered_map<item_id_t, Item *> map_item;
3     string txn_str;
4     num_txns = 0;
5     while (getline(ifs, txn_str)) {
6         istringstream iss(txn_str);
7         item_id_t item_id;
8         while (iss >> item_id) {
9             if (!map_item[item_id]) {
10                 map_item[item_id] = new Item(item_id);
11             }
12             map_item[item_id]->add_transaction(num_txns);
13         }
14         num_txns++;
15     }
16     vector<Item *> vec_items;
17     vec_items.reserve(map_item.size());
18     for (auto &item : map_item) {
19         vec_items.push_back(item.second);
20     }
21     return vec_items;
22 }
```

Nothing special here. Reads items from input file stream **ifs**, and return it.

I decided not to store transactions directly but store transaction ids in items (**line 12**). Scanning the whole database is much less efficient than my scheme.

For speedup, function uses **unordered_map** internally.

Find Frequent Patterns using Apriori Algorithm

```
1 vector<ItemSet *> find_frequent_patterns(vector<Item *> &items, const int min_support) {
2
3     auto freq_end = remove_if(items.begin(),
4                               items.end(),
5                               [min_support](auto &i) { return i->support() < min_support; });
6
7     vector<ItemSet *> freq_item_sets;
8     map<vector<item_id_t>, ItemSet *> candidate_item_sets;
9     for (auto it = items.begin(); it != freq_end; it++) {
10         freq_item_sets.push_back(*it);
11     }
12
13     size_t prev_begin = 0;
14     for (int k = 2; prev_begin != freq_item_sets.size(); k++) {
15         // Generate
16         for (auto i = prev_begin; i != freq_item_sets.size(); i++) {
17             for (auto j = i + 1; j != freq_item_sets.size(); j++) {
18                 auto &ii = freq_item_sets[i];
19                 auto &jj = freq_item_sets[j];
20                 auto uni = Item::get_union(ii, jj);
21                 if (uni->size() == k) {
22                     auto &uni_pos = candidate_item_sets[uni->items];
23                     if (uni_pos == nullptr) {
24                         uni_pos = uni;
25                     } else {
26                         uni_pos->add_child(ii);
27                         uni_pos->add_child(jj);
28                     }
29                 } else {
30                     delete uni;
31                 }
32             }
33         }
34         // Test
35         prev_begin = freq_item_sets.size();
36         for (auto &i : candidate_item_sets) {
37             if (i.second->num_children() == k && i.second->support() >= min_support) {
38                 freq_item_sets.push_back(i.second);
39             } else {
40                 delete i.second;
41             }
42         }
43         candidate_item_sets.clear();
44     }
45
46     return freq_item_sets;
47 }
```

Firstly, make rooms for frequent and candidate item sets. Items whose support is greater or equal than minimum support becomes frequent itemsets. (**line 3-11**)

and then do candidate generation and testing while frequent itemsets of size **k-1** exist. (**line 14**)

Generate. Generate candidate itemsets of size k via self-joining. Call `ItemSet::get_union()` to unify two itemsets, and test if resultant itemset is size k (line 18–21). If so, add it to candidate itemsets (line 22–27).

Test. For all candidate itemsets, check if all children are in frequent itemsets, and support is equal or greater than minimum support. If so, it becomes frequent itemset (line 36–42). Note that the number of children is k *if and only if* all children are in frequent itemsets.

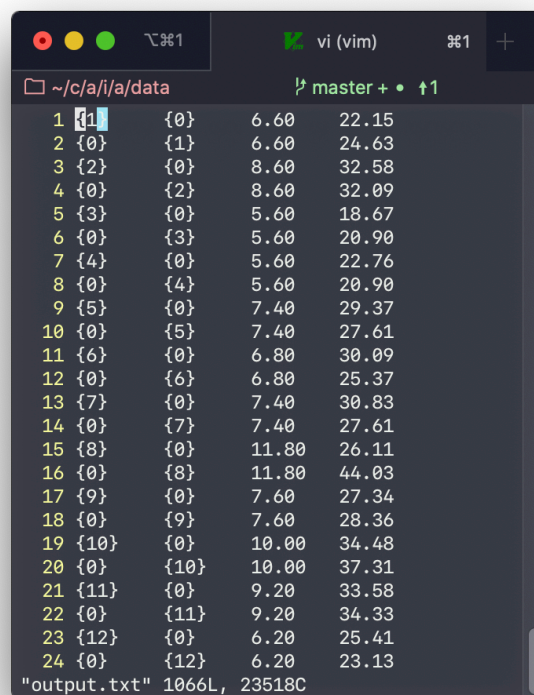
Find Association Rules

```
1 void find_association_rules(ofstream &ofs, vector<Item *> &items, const int min_support) {
2     auto freq_patterns = find_frequent_patterns(items, min_support);
3
4     for (auto &pattern : freq_patterns) {
5         auto descendants = pattern->get_descendants();
6         for (auto &descendant : descendants) {
7             ofs << AssociationRule(descendant, pattern);
8             num_assc_rules++;
9         }
10    }
11 }
```

Finding association rules can be done with descendants traversal.
For each descendant in each frequent itemsets, always one association rule is made.

Result

My program is able to find 1066 association rules of `data/input.txt` within 20~25ms (including I/O).



1	{0}	{0}	6.60	22.15
2	{0}	{1}	6.60	24.63
3	{2}	{0}	8.60	32.58
4	{0}	{2}	8.60	32.09
5	{3}	{0}	5.60	18.67
6	{0}	{3}	5.60	20.90
7	{4}	{0}	5.60	22.76
8	{0}	{4}	5.60	20.90
9	{5}	{0}	7.40	29.37
10	{0}	{5}	7.40	27.61
11	{6}	{0}	6.80	30.09
12	{0}	{6}	6.80	25.37
13	{7}	{0}	7.40	30.83
14	{0}	{7}	7.40	27.61
15	{8}	{0}	11.80	26.11
16	{0}	{8}	11.80	44.03
17	{9}	{0}	7.60	27.34
18	{0}	{9}	7.60	28.36
19	{10}	{0}	10.00	34.48
20	{0}	{10}	10.00	37.31
21	{11}	{0}	9.20	33.58
22	{0}	{11}	9.20	34.33
23	{12}	{0}	6.20	25.41
24	{0}	{12}	6.20	23.13

"output.txt" 1066L, 23518C