Advanced Data Structures

Creating a Data Structures Library in Golang

Authors:
Pol Alvarez Vecino

Professor:
Amalia Duch

June 15th 2018, Barcelona

Abstract

The goal of this project is to create a ready-to-use Github repository of advanced data structures (ADS) for Golang (Go). Go is a relatively young language and, despite having many libraries, it is still lacking in terms of available advanced data structures.

The language is statically typed and inherits a lot from C. However, it has Garbage Collection (GC) making it interesting (and easier) to implement some data structures (like skip lists) because we need not to remove orphan structures.

Furthermore, Go has integrated testing. This allows most IDEs to highlight regions which regions of the code are not being covered by the tests. This seems specially useful to find which corner cases of a data structure are not covered.

Contents

1 Introduction **3**

2 Union-find Sets **3**

2.1 Description 3

2.2 Variants 4

2.2.1 Union-by-rank 4

2.2.2 Path compression 5

2.3 Examples 6

2.3.1 Minimum Spanning Trees: Kruskal 6

2.3.2 Connected Components 6

3 Tries **7**

3.1 Description 7

3.2 Examples 8

3.2.1 Codeforces: A lot of games 8

4 Patricia Tries **9**

4.1 Description 9

5 Skip lists **10**

5.1 Description 10

5.2 Experiments 12

6 Frequent-Itemsets **13**

6.1 Item List 14

6.2 FIS Trie 14

6.3 FIS Patricia Trie 15

6.4 Experiments 15

7 Conclusions **16**

8 Future Work **16**

Appendices **17**

A Code forces problem D of contest 456: A lot of Games **17**

B Jutge problem: Connected Components **17**

C Using and running the library **18**

D String Representations **18**

1 Introduction

The goal of this project is to create as data structures repository for Go language [1]. The library can be found at

`https://github.com/kafkasl/golang-ads`

Specifically, the repository currently contains four ready-to-use standadlon advanced data structures (ADS) and three more applied to frequent itemsets mining (FIS):

- Advanced Data Structures:
  - Union-find Set
  - Trie
  - Patricia Trie
  - Skip-list
- Applications of ADS to Frequent Itemsets:
  - Item lists
  - FIS Trie
  - FIS Patricia Trie

All results, implementations, and examples have been coded from scratch in Go and each structure has an extensive test suite (labeled *data-structure-test.go* which covers most of the possible usages. Furthermore, understanding new ADS may be challenging at first so every implemented structure defines a *String*() method which gives a visual representation of it (for some structures, this methods are harder to implement than the actual structure). Section D provides a brief overview and example of each visual representation.

Additionally, *Union-find* and *Trie* structures come with an extra file (*example_test.go*) with examples of how to use it. For Union-find, the examples are an implementation of Kruskal's [1] minimum-spanning-tree and a competitive programming problem related to connected components (more details on subsection 2.3). For tries, the example is a competitive programming game described on 3.2.1.

Both examples have been implemented using the data structure as a library. However, the actual submissions of the problems for evaluation are under folder *./submission* because they need to be in a single file and read from standard input.

All tests and examples can easily run with the command: *go test* (for further details refer to Annex C).

Finally, folder *./evaluation* contains the performance/complexity tests of skip lists (see 5.2) and the memory evaluation of the three ADS used for FIS mining (see 6.4).

2 Union-find Sets

2.1 Description

Associated files:

./structures/union-find/

union-find.go, vector-based main UF implementation with path compression.

[1] https://golang.org/project/

union-find-rank-pointers.go, alternative UF implementation using rank and pointers.

union-find_test.go, test suite of UF path compression implementation.

union-find-rank-ptr_test.go, test suite of UF set rank implementation.

example_test.go, Kruskal's algorithm and connected components examples using UF set with path compression

./submissions/

submissionConnectedComponents.go, single-file implementation of the Jutge problem + stripped down UF set (as submitted to pass public/private tests and performance requirements).

Union-find sets are an easy to implement data structure but they are quite powerful. The goal of this sets (also called Disjoint-set or merge-find sets) is to hold many disjoint sets and efficiently report whether two elements belong to the same group. They usually have two operations: union and find (no big surprise there). As the name suggests, $union(x, y)$ takes two sets and joins them. On the other hand, $find(x)$ returns the ancestor of element x. We will describe the concept of ancestor next, but basically, two elements are in the same set if they have the same ancestor.

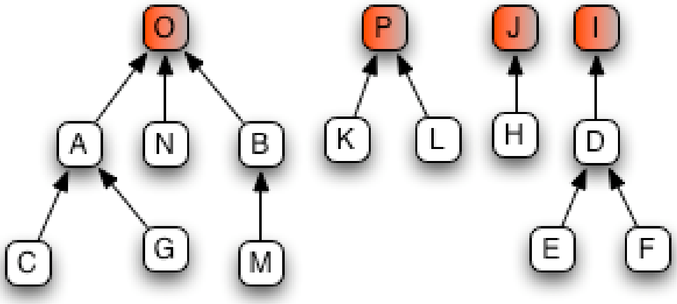


Figure 1: Three disjoint UF sets. Nodes marked red are the ancestors to be compared.

Figure 1 shows an example of UF set. Each time we call the $find(x)$ method we will climb the tree until we reach the ancestor (red nodes). If the ancestor of two nodes is the same, then they belong to the same set.

If the $join(x, y)$ operation is not correctly implemented we could end up with trees where height becomes $\mathcal{O}(n)$ (e.g. when always using the same node as parent). There are different ways of avoiding this. In this project we implemented two of them: Union-by-rank and path compression.

2.2 Variants

2.2.1 Union-by-rank

The first implementation uses pointers to keep track of ancestors and a *rank* value which keeps the trees of the sets balanced. Each set is a triplet $(parent, rank, value)$ where parent is a pointer to the node's parent and is nil for single-item sets, the rank is a value used during union to decide which elements becomes parent of the other, value is the number representing the node.

Initially, all nodes have rank 0. When joining two sets, if they have the same rank, one of them gets its rank incremented by 1 and becomes the parent of the other. If one of them has higher rank, it becomes the parent of the other directly without rank updates.

Using this implementation, the height of a set is never going to be greater than $log(n)$ so the the $find(x)$ operation has a worst case cost of $\mathcal{O}(logn)$ (when the node x is a leaf) and $union(x, y)$ has cost $\mathcal{O}(1)$ (just linking the pointer to a new parent).

2.2.2 Path compression

The second implementation is based on flattening the trees to height 1 by compressing the path of a given node. It is implemented with a vector. Each position of the vector contains the position of the node's parent (or one of its ancestors thanks to path compression). In order to compress the paths, whenever the $find(x)$ operation is called, we use the chance to update the ancestor of the node to make it point directly to the root of the tree (instead of pointing to a parent or non-root node) as follows:

```
Find(node int):
    if parent(node) == node:
        return x

    # we update the pointer to the ancestor recursively
    parent(node) = Find(parent(node)

    return parent(node)
```

The union operation is just the result of making one of the ancestors point to the other (arbitrarily because path compression takes care of the balancing).

The amortized cost of the operations using path compression is described in [2]. They define a potential function using Ackermann's inverse to prove that the amortized costs are:

$$find(x)\mathcal{O}(\alpha(n)$$

$$union(x, y)\mathcal{O}(\alpha(n)$$

where $\alpha(x)$ is Ackermann's inverse that grows extremely slowly,

$$\alpha(x) = min\{k : \mathcal{A}_k(1) \geq x\}$$

$$\mathcal{A}_k(j) = \begin{cases} j + 1, & \text{if } k = 0. \\ \mathcal{A}_{k-1}^{(j+1)}(j), & \text{if } k \geq 1. \end{cases}$$

Path compression costs are better than Union-by-rank. However, the worst case cost of $find(x)$ in path compression is $\mathcal{O}(n)$, when we have a tree with height n instead of the $log(n)$ of Union-by-rank. Despite that, in most scenarios the implementation details end up making Union-by-rank slower and it is rare to find a use case where we want to strictly ensure that worst case is bound because the amortized cost is not good enough.

2.3 Examples

2.3.1 Minimum Spanning Trees: Kruskal

One of the best well-known examples for union-find sets is to implement Kruskal's minimum spanning tree (mst) algorithm. In order to find it, Kruskal's idea is to first order all edges by their weight. Then add edges greedily to the tree if they are not already both part of the same partition. To do so with the UF sets, we initialize each node as a UF set. Then, each time we need to check if two nodes are in the same set, we compare their ”ancestor” with $find(x)$ function. If they are not, we join them and repeat the process until the mst is built.

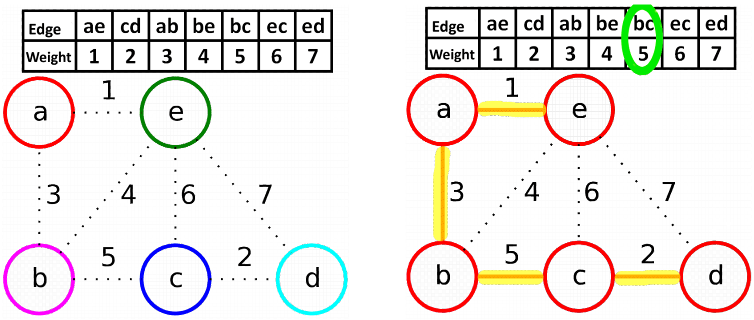
The example's implementation is divided into three calls:

parseInput(input) takes triplets of $(origin\_node, destiny\_node, weight)$ and returns and adjacency matrix.

adjMatrixToPriorityQ(adjMatrix) takes the adjacency matrix and builds the priority queue of edges ordered by weight.

kruskal(priorityQ) takes the priority Q of edges and returns list of triplets $(origin\_node, destiny\_node, weight)$ that conform the minimum spanning tree.

Figure 2 shows the visual representation of the example in the repository. First, edges are ordered (top-left) and they are added one by one. Note that vertex $be : 4$ is not added because it would create a cycle $a \to e \to b \to a$ (i.e. node e and b are in the same union-find set).



Edge	ae	cd	ab	be	bc	ec	ed
Weight	1	2	3	4	5	6	7

Edge	ae	cd	ab	be	bc	ec	ed
Weight	1	2	3	4	5	6	7

Figure 2: (left) Weighted graph and its edges ordered by weight. (right) Minimum spanning tree obtained applying Kruskal's algorithm to left image. In green last edge added to the mst (image extracted from Wikipedia's MST entry)

2.3.2 Connected Components

This example is based on the Jutge programming problem described in Appendix B. The goal is to count how many connected components does a graph $G = (V, E)$ have after inserting k edges into it and starting with 0 edges.

To solve this problem, we know that at the beginning (when no edges exist) the $\#connected\_components = |V|$. As in Kruskal's example, we initialize every vertex as a different UF set. Then, for each new vertex to add, we check if the vertex's nodes are in the UF set. If they are, then number of connected components remains the same. Otherwise, the number connected components decreases by one (two different sets/graphs are joined) and we join them. The number of connected components at each step is $|V|-$ # times two nodes were disjoint.

3 Tries

3.1 Description

Associated files:

./structures/trie/

trie.go, trie implementation.

trie_test.go, test suite of trie implementation.

example_test.go, example of a Codeforces [2] game solved using tries.

./submissions/

submission456D.go, single-file implementation of the Codeforces problem + stripped down trie (as submitted to pass public/private tests and performance requirements).

Tries are an ordered search tree structure where children of a node are indexed by part of the value of an element. Frequently, they are used with strings and children are indexed by each letter of the word, or numbers indexed by some radix that decomposes the number into digits. From now on, we assume we are dealing with string/letter tries without loss of generality.

In a trie, each node contains a dictionary or an array of children indexed by the keys, and a boolean indicating if some word finishes on the node (called *endOfWord* from here onwards) [3]. Figure 3 shows an example of an string trie.

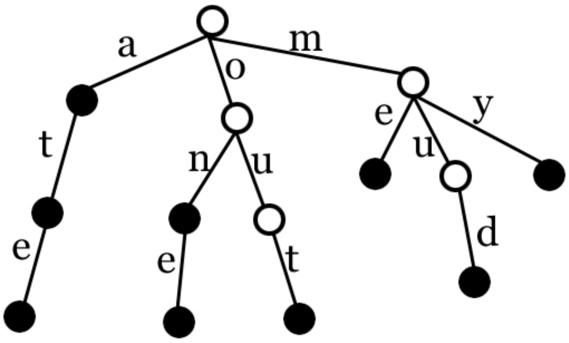


Figure 3: Example of a string trie after inserting words: a, at, ate, on, one, out, my, me, mud. Notice that black nodes indicate that the boolean *endOfWord* is true. Thus ”a”, ”at”, ”on” are all present in the trie, but ”ou” and ”mu” are not.

To check if an element is in the trie, we traverse the tree following at each node the child corresponding to the next letter of the string. If we arrive to a node with $endOfWord = true$ then the element is contained on the list. If we reach a null child (the child associated with the next key does not exist) or the final node has $endOfWord = false$, then element is not contained in the trie.

```
search(word):
    currentNode <- root(Trie)


    for each letter in word
```

[2] codeforces.com

[3] There are many trie implementations and some of them store in special nodes dangling from their corresponding search path. In this project, the element is its search path itself and the trie has no special nodes, just the bool indicating if a sequence finishes in a given node.

```
        if children(currentNode) contain letter
            currentNode <- children indexed by the letter
        else
            # next letter does not point to a node -> word is not present
            return false

    # we have seen all letters of the word
    # if current node marks the end of a word, then our word was present

    return isEndOfAWord(currentNode)
```

To insert an element on the trie we just need to follow the search procedure up to a given node n. If we have traversed all the word, we just mark the node with $endOfWord = true$. Otherwise, for each remaining letter, we create a new node and make the last node point to it using the letter as index.

```
insert(Trie, word):
    currentNode <- root(Trie)

    for each letter in word
        if children(currentNode) contain letter
            currentNode <- children indexed by the letter
        else
            newNode <- create new node
            add letter to children(currentNode) as pointer to newNode
            currentNode <- newNode

    # mark the currentNode as being the end of a word
currentNode.endOfWord = true
```

As a side note, when using an array representation, we need to know beforehand the length of our alphabet (our resize the array accordingly when new keys are added). On the other hand, in our implementation the children are indexed by a map of runes [4] keys, effectively allowing to use any of the 137,000 unicode characters as key.

3.2 Examples

3.2.1 Codeforces: A lot of games

The example for tries is a problem from codeforces (see Appendix A). It simulates a game in which two players are given a set of words. They have to construct and string together such that it is a prefix of a least one of the words. Each player must add a letter each turn. If a player can not construct such a string in its turn loses. The game has n rounds, and the loser of game $k-1$ starts playing game k. The winner of the game is the winner of the last round so it may be convenient to lose some rounds to start the next ones.

The strategy is to build a trie with all the available words and explore it recursively. Basically, the first nodes of the trie correspond to the first player's possible moves. The children of these nodes correspond to second player's, and so on. In each turn, we want to know if it is our decision to win or lose (we are able to win/lose

[4] runes are a Go type which represent unicode characters (under-the-hood they are a int32)

regardless of the other player's movements). Expressed in a recursive fashion, at a certain step, we can win if the other player can not win with the children moves and vice-versa.

This is the pseudocode for the recursive function

```
canWinLose(TrieNode):
if children(TrieNode) is empty
        # can not win (no moves left)
        # can lose
        return canWin=False, canLose=True

    else
        initialize canWin and canLose to false
        for each child of children(TrieNode)
            child_canWin child_canLose = canWinLose(child)
            # canWin if I could already win, or if child can not win with child movement
            canWin |= !child_canWin
            # canLose if I could already lose, or if child can not lose with child movement
            canLose |= !child_canLose

        return canWin, canLose
```

Then, if the first player can win and lose any round when he starts, he will win the game (he will win/lose accordingly to start the last round). If he can only win when starting, then he will win when the number of rounds is odd (e.g. can win 1st round, lose 2nd, and win the 3rd), and the second player will win when the number of rounds is even. If first player can neither win nor lose when starting, then player 2 will win the game always.

4 Patricia Tries

4.1 Description

Associated files:

./structures/patricia-trie/

patricia-trie.go, patricia trie implementation based on Handbook of DS and Applications.

patricia-trie_test.go, test suite of patricia trie implementation.

example_test.go, replication of the toy example found in Patricia Trie section of Handbook of Data Structures and Applications book.

Patricia (Practical Algorithm To Retrieve Information Coded In Alphanumeric) tries [3] are compressed tries [4] with the values contained in the nodes themselves.

In compressed tries, instead of branching at each element (e.g. each letter of a word) we have an additional field *skip* which indicates which is the next different element. The basic advantage with respect to compressed tries is that Patricia tries meld element and search nodes into one.

Figure 4 shows a compressed trie and its associated Patricia version.

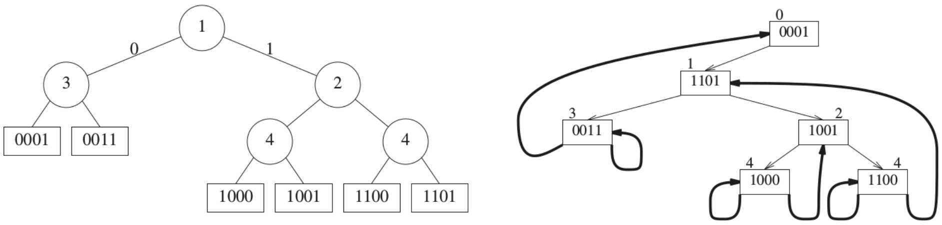


Figure 4: (left) Compressed trie. (right) Patricia trie to left image. (image extracted from [5])

To find an element in a Patricia trie, we follow the path the same way as in compressed tries. However, instead of reaching an element node (a leaf) or null value, we will reach a node x with a *skip* value inferior to the previous node. If the value associated to node x is the one we were looking for, then the element is present. Otherwise the element is not present in the trie.

This implementation of the Patricia tries follows strictly the description found in Handbook of Data Structures and Applications [5]. We do not elaborate further on the insert mechanism because Patricia tries start with an special node *header* making the insertion a bit cumbersome because we have to deal with many cases (when no elements exist, when just the header exists, and when the header and more elements have been inserted).

5 Skip lists

5.1 Description

Associated files:

./structures/skip-list/

skip-list.go, full skip list implementation.

skip-list_test.go, test suite of skip lists implementation.

example_test.go, toy example of how to create a skip list and print it.

./evaluation/

skip-list_performance.go, evaluation of skip lists expected path, size, and height.

Skip lists [6] are a randomized search structure for ordered elements. They were invented as an alternatively to balanced trees. The randomization introduced makes the search, insert, and delete operations have a very good expected performance regardless of the input.

A skip list $\mathcal{S}$ for a set $\mathcal{X}$ is formed by a list of lists $\mathcal{L}_i$. $\mathcal{L}_1$ is called level 1 list and contains all the elements. For $\mathcal{L}_i$ with $i > 1$, if an element x belongs to $\mathcal{L}_{i-1}$ then it belongs to $\mathcal{L}_i$ with probability q s.t. $0 < q < 1$. The actual parameter set in a skip list is p and $q = 1 - p$.

The information is stored in a collection of nodes. Each node represents a value and contains a list of pointers to its successor nodes at each level. First and last node (header and NIL from here onwards) are always present and act as limits with values $-\infty$ and $+\infty$ respectively.

Figure 5 shows a skip list built with $p = 0.5$. The given height of a node is the length of its successor's list (e.g. $height(37) = 3$ or $height(53) = 4$). The probability that a given node x has height k is given by the geometric random variable:

$$P[height(x) = k] = p * k^{k-1}$$

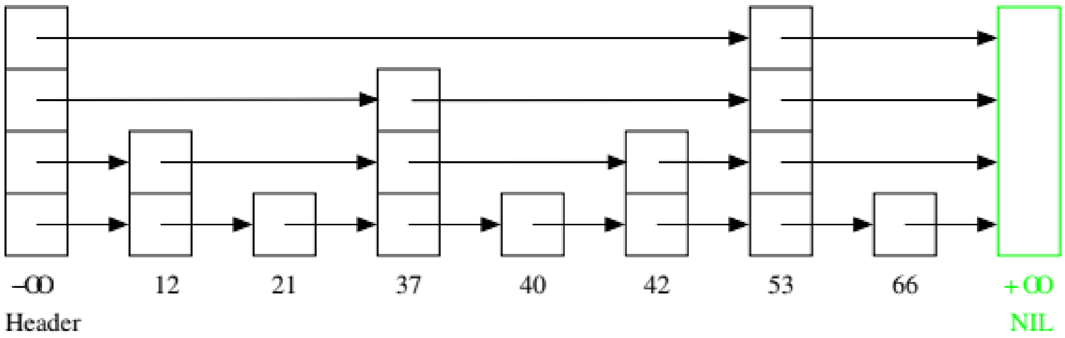


Figure 5: Example of a skip list of height 4. (image extracted from Conrado's Martínez PWL slides).

The height of the skip list itself is the highest node height.

$$height(\mathcal{S}) = max(height(x)), \forall x \in \mathcal{S}$$

To search a key x in a skip list the procedure is as follows: we start at the header and level $l = height(\mathcal{S})$. If the key of the successor at current level is bigger than x, we get down a level $l = l - 1$. Otherwise, we follow the link to the successor in the current level. Finally, when $l = -1$, the successor at level 0 must hold the key. If the successor is either NIL (last node) or the successor's key is not x's key then it is not present in $\mathcal{S}$. In pseudocode;

```
search(x)
    currentNode <- header
    l <- height(S)
    while l >= 0
        if successor(currentNode) at level l is NIL or the sucessor's key is <= x
            # get down a level
            l <- l - 1
        else
            # move to succesor
            currentNode = successor(currentNode) a level l
    if successor(currentNode) at level l is NIL or the sucessor's key is not x
        # key is not present
        return nil
    else
        # successor key is x
        return successor(currentNode)
```

To insert an element, we follow the same procedure as the search. If we find the key, we just update the value. If not, we insert a new node between the *currentNode* and its successor. Additionally, we need to keep track of the predecessors at each level to update their pointers. The height of the new node is computed using the q parameter as stated earlier. Deleting a node is almost identical: when we reach the node to be removed we just rewire the predecessors that pointed to the deleted node to its successors.

5.2 Experiments

We now introduce the concept **length of a path** in order to evaluate the complexity of the operations in a skip list. The length of path is the number of steps required from the header's highest level until we find the desired node (or confirm that is not present). It is basically the number of times we jump to a successor plus the times to get down one level. The expected path length of a search is given by:

$$E[length(path)] \leq \frac{1}{q} * \log_{1/q} n + \frac{1}{p} \tag{1}$$

The expected number of pointers per node is $1/p$ so when p is close to 1 we have flatter skip lists. Alternatively, when p is closer to 0, nodes become taller. Pugh, the author of the original paper, suggested using $p = 3/4$ as value for a good trade-off between space and time.

Figures 6 and 7 show the average skip list height, search path length, and memory footprint when varying the parameter p. Twenty tests were executed or each value of parameter p and the mean values are reported. The inputs and searches were lists of randomly generated key of 10.000 and 100.000 elements respectively.

As expected, the closer p gets to 1, the smaller the memory footprint and average height. At first, path length also seems to decrease until an inflection point around 0.7 where it starts to increase. This point represents the best trade-off between time efficiency (small path length) and space efficiency (shorter nodes).

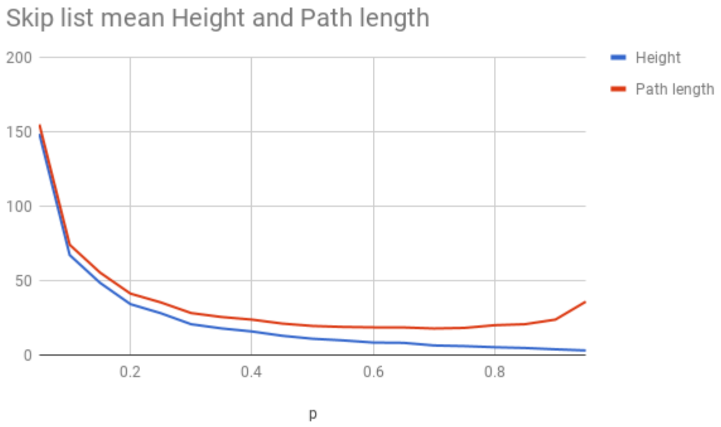


Figure 6: Mean path length vs. skip list size for different values of parameter p.

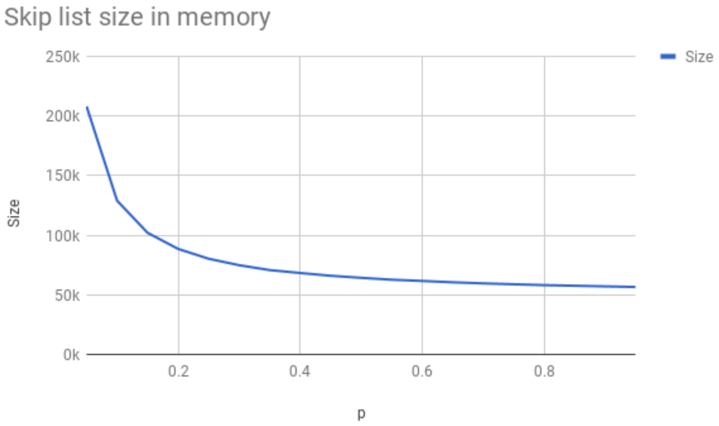


Figure 7: Memory footprint of an skip list for different values of parameter p.

Figure 8 shows the empirical mean height with respect to its theoretical upper bound computed as in equation 1. We observe that the bound is a lower than the mean for small values of p and becomes bigger than it from $p \geq 0.6$ approximately. This could be caused by some hidden constant or maybe some implementation detail.

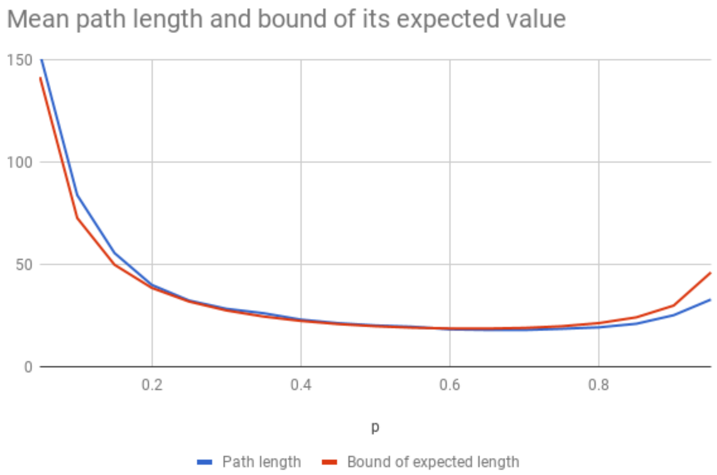


Figure 8: Mean path length vs. the upper bound of the expected path length skip list for different values of parameter p.

6 Frequent-Itemsets

Associated files:

./structures/frequent-itemsets/

item-list.go, item list basic representation.

fis-trie.go, frequent itemsets trie structure implementation.

fis-patricia.go, frequent itemsets patricia trie representation structure.

fis_tests.go, test suite of frequent itemsets structures.

example_test.go, demos on how to generate the examples presented in [7].

./evaluation/

fis_size-tests.go, code used to generate the experiments results of subsection 6.4.

Finding the Frequent Itemsets (FIS) of a record of transactions is a Data Mining problem. Given a list of transactions $\mathcal{T}$, each transaction t is a variable-size subset of all available items $\mathcal{E}$. The goal is to find the sets of items which occur together most frequently in all transactions. The number of occurrences of a set $\mathcal{X}$ called support of the set $supp(\mathcal{X})$.

A common transaction's representation is as list of lists of numbers (each number represents a different element). This structure can be quite big and the naive counting approach to find the frequent itemsets can be impractically slow. This section implements part of the work found in [7].

In the paper, authors come up with an algorithm to mine frequent-sets efficiently. This project only implements the following data structures: Item List, Trie, Patricia Trie; and part of the comparison of their final sizes (Table 1, using Item lists instead of arrays). The actual results differ because we evaluate the size of the structures with a function that reports the actual data structure size in memory (instead of defining some parameters as they do).

Usually, when mining itemsets, a minimum support threshold is defined in order to remove elements with less occurrences. The descriptions of how to filter by minimum threshold during the construction of the data structures are not detailed because in our case its done during the first item list creation and it is straightforward.

6.1 Item List

Item List is the first representation of the transactions dataset. It is composed of a list of tuples *(element, support, reference)* ordered by their support. The reference points to a threaded list. Each threaded list is formed by a transaction (shared by all lists) and a pointer to the next threaded list where the *element* associated with a given tuple appears. Figure 9 shows an example of a dataset an its associated item list.

Once the Item List is built, the transactions' elements are sorted according to their support. This data structure is not space efficient, it is a first step in order to build the trie representation of the dataset. The expect size of an item list is the average mean transaction length (n) times the number of transactions (t) plus $3 * t$ (to store the tuples). Given an alphabet of m elements the bound of the space (s) is:

$O(s) = n * t + 3 * m$

TID	Items
1	**A B D** E **F G H** I
2	**B** C E **L**
3	**A B D F H L**
4	**A B** C **D F G L**
5	**B G H L**
6	**A B D F** I

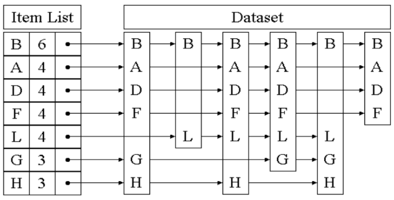


Figure 9: Sample dataset (bold items are frequent for a minimum support of 3) and its associated Item List. (image extracted from [7])

6.2 FIS Trie

The trie used for frequent itemsets is similar to the one described in Section 3. The basic difference is that nodes now contain the support associated to the itemset formed by the path up to the node. Also, they no longer contain the $endOfWord$ boolean nor the $search()$ function because those operations make no sense in this context.

Figure 10 shows the trie that results of the sample dataset in Figure 9 and a minimum support of 3. For example, the prefix of node $D : 4$ is BAD indicating that $sup(BAD) = 4$.

The main advantage of the trie representation is that all common prefixes will not be duplicated and thus it will avoid information duplication while retaining the frequent itemsets' support. However, sparse datasets (where the ordered transactions share very small prefixes) will not be as space efficient because they will have many repeated nodes depending on which transaction they belong to. The next subsection describes how to improve the space complexity by using Patricia tries.

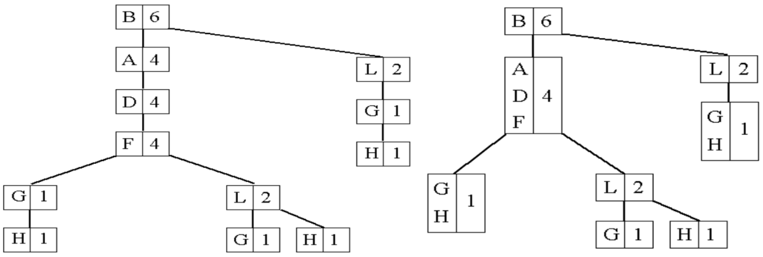


Figure 10: Frequent Itemset trie and Patricia trie for sample dataset from Figure 9 (image extracted from [7])

6.3 FIS Patricia Trie

The Patricia trie used for frequent itemsets mining is basically an ad-hoc implementation built from compressing a FIS trie structure. The idea is simple, all nodes with a single child and the same count need are part of the same frequent itemset and thus can be merged into a single node (avoiding unnecessary memory allocation for extra pointers, and duplicated counts).

Figure 10 shows the Patricia trie that results of the sample dataset in Figure 9 and a minimum support of 3.

6.4 Experiments

Figure 1 shows the size of each structure and each dataset [5]. As stated earlier, the array present in the paper is substituted here by the Item List baseline. The memory size is computed using "OneOfOne's" go-utils module [6].

We observe that depending on the sparsity of the transaction dataset, the compression achieved by patricia tries ranges from a small percentage (5%) to more than half the space. Albeit being computed differently, the compression rates between trie and patricia trie remain the same as in the original paper.

The comparison code is placed in *./evaluation* folder instead of as an example because it takes longer than the maximum allowed for tests/examples.

Table 1: Comparison of the size in memory of different datasets when represented using an Item List, a Trie, or Patricia Trie.

	Absolute sizes			Relative sizes		
Dataset \Structure	Item List	Trie	Patricia Trie	Item List	Trie	Patricia Trie
Chess	3.020.040	1.590.589	589.908	100%	52%	19%
Mushroom	4.944.232	1.133.163	960.827	100%	23%	19%
Pumsb*	62.234.208	38.104.128	16.697.927	100%	61%	27%
Pumsb	89.918.176	48.757.701	27.863.724	100%	54%	31%
Connect	73.507.256	14.873.604	10.656.609	100%	20%	14%

[5] Datasets can be found at: `http://fimi.ua.ac.be/data/`

[6] `https://github.com/OneOfOne/go-utils/`

7 Conclusions

In the first part of this project, we have implemented four data structures in Python. For each structure, we have provided a test coverage of over 90%. Most of them have been applied to either competitive programming problems or usual applications showing their efficiency. In order to ease their usage, *stringify* methods have been implemented for the most demanding ones like trees or skip lists.

For skip lists, we have evaluated their expected performance. The results obtained show that their complexity (represented by their path length) is indeed inside the theoretical bounds. We have also seen the approximate point at which the trade-off between space and time complexity is optimal (around $p = 0.75$).

For the second part, we have applied some of the implemented structures to a real world problem: mining frequent itemsets. The work done is mostly structural (i.e. representing the data with a given structure, not using it). As such, the comparisons performed are related to the space complexity each of the structures requires. Specifically, we have taken three structures - *Item list, Trie, Patricia Trie* - and 5 common data mining datasets and measured the memory space they use.

The results show what we expected: the item list (which contains always all transactions) is always the biggest one; the trie, which is able to compress common prefixes, improves a lot the space needed; and finally, the Patricia trie which is able to compress nodes, reduces the trie space even further, specially in sparse datasets.

As a closing note, I would like to mention that Golang integrated testing (which highlights the paths not being tested) has been a highly valuable asset during the implementations. Being able to tell which execution paths of your data structure are not being tested greatly enhances one's ability to design test suites and find the corner cases. More than once, a structure which seemed perfectly fine, failed miserably when untested parts of its code were included in the test cases.

8 Future Work

Future work is concerned basically on adding more structures to the library. A worthy next candidate structure is the Binary Indexed Tree.

Improving the documentation inside the source files is also a must. The source files are quite lacking and understanding other's implementations of complex data structures (looking at you Patricia) may be challenging without well documented code (specially when implementations vary depending on the source material).

Recently I have come across a repository with the same goal [7]. Given that the repository is older and has more contributors and maturity, the next goal will be to migrate the implementation to that repository (which does not contain a single one of the implemented ones here).

References

[1] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, p. 48, 1956.

[2] G. C. Harfst and E. M. Reingold, "A Potential-Based Amortized Analysis of th e Union-Find Data Structur e,"

[3] D. E. Knuth and D. E., *The art of computer programming vol. 3.* Redwood City: Addison-Wesley Pub. Co, 1998.

[7] `https://github.com/emirpasic/gods`

[4] K. Maly and Kurt, “Compressed tries,” *Communications of the ACM*, vol. 19, pp. 409–415, jul 1976.

[5] Mehta and Sahni, *Handbook Of Data Structures And Applications*, vol. 20043742 of *Chapman & Hall/CRC Computer & Information Science Series*. Chapman and Hall/CRC, oct 2005.

[6] W. Pugh and William, “Skip lists: a probabilistic alternative to balanced trees,” *Communications of the ACM*, vol. 33, pp. 668–676, jun 1990.

[7] A. Pietracaprina and D. Zandolin, “Mining frequent itemsets using patricia tries,” in *CEUR Workshop Proceedings*, vol. 90, 2003.

Appendices

A Code forces problem D of contest 456: A lot of Games

Full problem here `http://codeforces.com/contest/456/problem/D`

> time limit per test1 second
>
> memory limit per test256 megabytes
>
> input: standard input
>
> output: standard output

Andrew, Fedor and Alex are inventive guys. Now they invent the game with strings for two players.

Given a group of n non-empty strings. During the game two players build the word together, initially the word is empty. The players move in turns. On his step player must add a single letter in the end of the word, the resulting word must be prefix of at least one string from the group. A player loses if he cannot move.

Andrew and Alex decided to play this game k times. The player who is the loser of the i-th game makes the first move in the $(i + 1)$-th game. Guys decided that the winner of all games is the player who wins the last $(k - th)$ game. Andrew and Alex already started the game. Fedor wants to know who wins the game if both players will play optimally. Help him.

Input The first line contains two integers, n and k ($1 \leq n \leq 105; 1 \leq k \leq 109$).

Each of the next n lines contains a single non-empty string from the given group. The total length of all strings from the group doesn’t exceed 105. Each string of the group consists only of lowercase English letters.

Output If the player who moves first wins, print ”First”, otherwise print ”Second” (without the quotes).

B Jutge problem: Connected Components

Full problem here: `https://jutge.org/problems/P94041_ca/statement` In an undirected graph with n nodes, and, initially without any vertex, we need to insert m vertices in the given order, stating how many connected components are there after each insertion.

Input

Input consists in many different cases. Each case starts with n and m, followed by the vertices to insert. Each vertex is formed by its two vertices. Suppose that $2 \leq n \leq 105; 1 \leq m \leq 2n$, vertices are numbered between 0 and $n - 1$, there are no repeated vertices, and no vertex connects a node with itself.

Output

For each case, write a line with m number space-separated. K-th element should be the number of connected components of the graph after inserting the first k vertices of the input.

C Using and running the library

Folder $/bin$ contains a script called $run_{t}ests_{a}nd_{e}xamples.sh$ which runs all the test suites and examples of all the data structures and frequent-itemsets use case.

In order to run a the test suite and examples of a single folder issue:

Listing 1: Run tests and examples of a of union-find in verbose mode (-v)

```
# Running all tests and examples of union−find
cd ./structures/union−find
go test −v

# Running a single example or test of a given data structure
# in this example TestFindAndUnion of union−find package
cd ./structures/union−find
go test −run TestFindAndUnion
```

D String Representations

This sections provides a brief description of the visual representation of each one of the implemented structures for easier usage.

Standard string trie's representation with words "malleus", "mars, and "me" is:

```
Trie:
 └─ m
     ├─ a
     │   ├─ l
     │   │   └─ l
     │   │       └─ e
     │   │           └─ u
     │   │               └─ s
     │   └─ r
     │       └─ s
     └─ e
```

Patricia tries are represented in a tree fashion. Each level is formed by $value[split] - > (lc, rc)$ where $value$ is the value of the node, $split$ is the index of the of the bit to be compared at the node, lc, rc are the pointer to the left and right child. If a pointer is _ it means it is a self-pointer.

```
Patricia Trie:
Header
└──  101[0] -> (1000, _)
     └── L 1000[61] -> (1010, _)
         ├── L 0[62] -> (100, _)
         │   ├── L 10[63] -> (10, _)
```

```
        │   └── R 100[64] -> (101, _)
        └── R 1010[63] -> (1010, _)
```

Union-find set with path compression's representation is just a vector where each position's number points to its ancestor (whether it is the ultimate ancestor or just a parent) or to itself if it is the root of a set. In the example, each pair of sets have been united (1 with 0, 3 with 2, etc...).

```
1 1 3 3 5 5 7 7 9 9
```

Union-find with rank representation is a bit more elaborated (because any node can be printed but every node only has a pointer to its parent, not children). It basically is a print from each node up to its ancestor. The shown representation is the result of joining 4 nodes. First we join 0-1 and 2-3, and then we join the resulting trees. $[1] : 1- > [2] : 3$ means node with height 2 and value 3 is ancestor of node with height 1 and value 1.

```
[0]: 0 -> [1]: 1 -> [2]: 3
[1]: 1 -> [2]: 3
[0]: 2 -> [2]: 3
[2]: 3‘
```

Skip lists are represented in a top-bottom fashion. On the right hand side, we find for each node $[key : values(height)]$. Each level has as many boxes as the height of the node, and the arrows point to the successor of each node at each level (being the level 0 leftmost one).

```
Skip List (height: 4)
[ ] [ ] [ ] [ ] [-∞ : start (4)]
 |   |   |   |
 V   V   V   V
[ ] [ ] [ ] [ ] [2: ab (4)]
 |   |   |   |
 V   V   |   |
[ ] [ ]  |   |  [4: ba (2)]
 |   |   |   |
[ ]  |   |   |  [5: bb (1)]
 |   |   |   |
 V   V   V   |
[ ] [ ] [ ]  |  [6: bc (3)]
 |   |   |   |
 V   |   |   |
[ ]  |   |   |  [9: cc (1)]
 |   |   |   |
 V   V   V   V
[ ] [ ] [ ] [ ] [+∞: end (1)]
```

Item list representation is a list of elements ordered by decreasing support. Each line contains the element, its support inside brackets and then all the transactions to where is belongs:

```
B [6]: [B A D F G H] + [B L] + [B A D F L H] + [B A D F L G] + [B L G H] + [B A D F]
A [4]: [B A D F G H] + [B A D F L H] + [B A D F L G] + [B A D F]
D [4]: [B A D F G H] + [B A D F L H] + [B A D F L G] + [B A D F]
F [4]: [B A D F G H] + [B A D F L H] + [B A D F L G] + [B A D F]
L [4]: [B L] + [B A D F L H] + [B A D F L G] + [B L G H]
G [3]: [B A D F G H] + [B A D F L G] + [B L G H]
```

```
H [3]: [B A D F G H] + [B A D F L H] + [B L G H]
```

Frequent itemset trie representation is the same as the regular trie but now each node has its accumulated support between brackets:

```
Trie:
 └─ B[6]
    ├─ A[4]
    │  └─ D[4]
    │     └─ F[4]
    │        ├─ G[1]
    │        │  └─ H[1]
    │        └─ L[2]
    │           ├─ G[1]
    │           └─ H[1]
    └─ L[2]
       └─ G[1]
          └─ H[1]
```

Frequent itemset patricia trie is almost identical with the FIS trie but nodes at a certain position i with skip value > 1 are represented concatenating the keys from i to $i + skip$.

```
Patricia trie:
 └─ B[6]
    ├─ ADF[4]
    │  ├─ GH[1]
    │  └─ L[2]
    │     ├─ G[1]
    │     └─ H[1]
    └─ L[2]
       └─ GH[1]
```