A $O(n)$ UNT ALGORITHM FOR THE UNION-FIND PROBLEM

Oscar Riveros

March 2, 2017

oscar.riveros@peqnp.com
http://twitter.com/maxtuno
http://klout.com/maxtuno
http://independent.academia.edu/oarr

1 THE UNION-FIND PROBLEM

The general setting for the union-find problem is that we are maintaining a collection of disjoint sets $\{S_0, S_1 \dots S_{k-1}\}$ over some universe, with the following operations:

`union(x, y)`: replace the set $\{x\}$ is in (let's call it S) and the set $\{y\}$ is in (let's call it $S^{'}$) with the single set $S \bigcup S^{'}$.

`connected(x, y)`: test if exist a subset S with x and $y \in S$.

2 DATA REPRESENTATION

Suppose there is an arbitrary but finite number of elements $\{x_0, x_1 \dots x_{k-1}\}$ then can represent the entire structure with, $\{1, 2, 4, \dots, 2^{(k-1)}\}$ and the subsets how $\{x_{i_0}, x_{i_1} \dots x_{i_l}\}$=$\sum_{i \in \{i_0, \dots, i_l\}} 2^i$.

Example 1. Let be $U = \{P, NP, =, \neq\}$ then the subset $\{P, =, NP\}$ is writting like $2^0 + 2^1 + 2^3 = 11$, and $\{P, \neq, NP\}$is writting like $2^0 + 2^1 + 2^4 = 19$.

3 ALGORITHM DESIGN & IMPLEMENTATION

With all generality can only work with integers forget the particular giben set, then implementation is as follows:

3.0.1 PYTHON IMPLEMENTATION

```
universe = []

def union(a, b):
    element = (1 << a) + (1 << b)
    idx = 0
    while idx != len(universe):
        if universe[idx] & element:
            universe[idx] |= element
            element = universe[idx]
            del universe[idx]
            idx -= 1
        idx += 1
    universe.append(element)

def connected(a, b):
    element = (1 << a) + (1 << b)
    for item in universe:
        if item & element:
            return item
    return 0
```

4 EXPLANATION

4.1 UNION

1. There is the simple unsorted list for storage the elements and fusion them

   ```
   universe = []
   ```

2. The definition of function, this receives two arguments that are the indices of the elements of the original set.

   ```
   def union(a, b):
   ```

3. This is equivalent to $2^a + 2^b$ but it is more efficient for that is a bit shifing and not an exponencicion

   ```
   element = (1 << a) + (1 << b)
   ```

4. It is to merge elements if necessary, the loop is executed $|U|$ times, that is equal to number of subsets or paths.

   ```
   idx = 0
   while idx != len(universe):
   ```

5. If there are elements in common the merges and eliminates the excess if not exit the loop, this operation is executed each time t

   ```
   if universe[idx] & element:
       universe[idx] |= element
       element = universe[idx]
       del universe[idx]
       idx -= 1
   idx += 1
   ```

6. Finally append the new element.

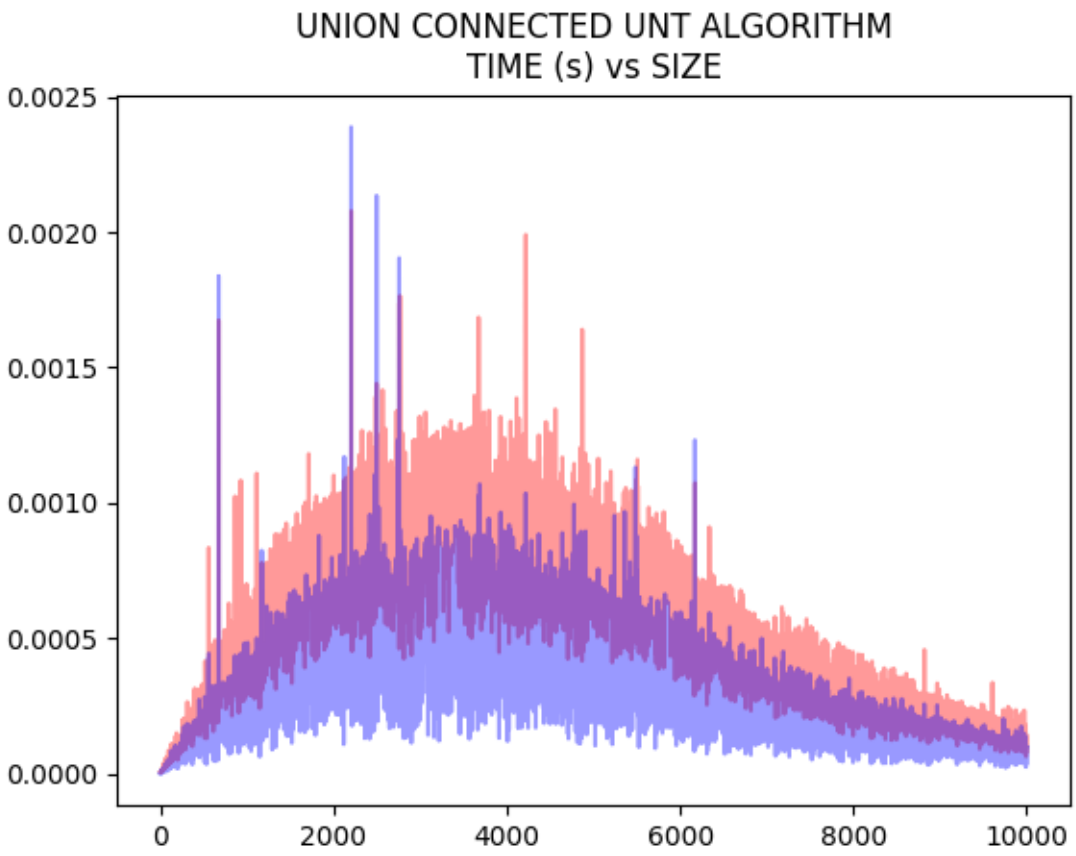
   ```
   universe.append(element)
   ```

4.2 CONNECTED

The explanation its the same, the diference is when it finds a match, it returns the subset, if not it returns 0 ($\emptyset$).

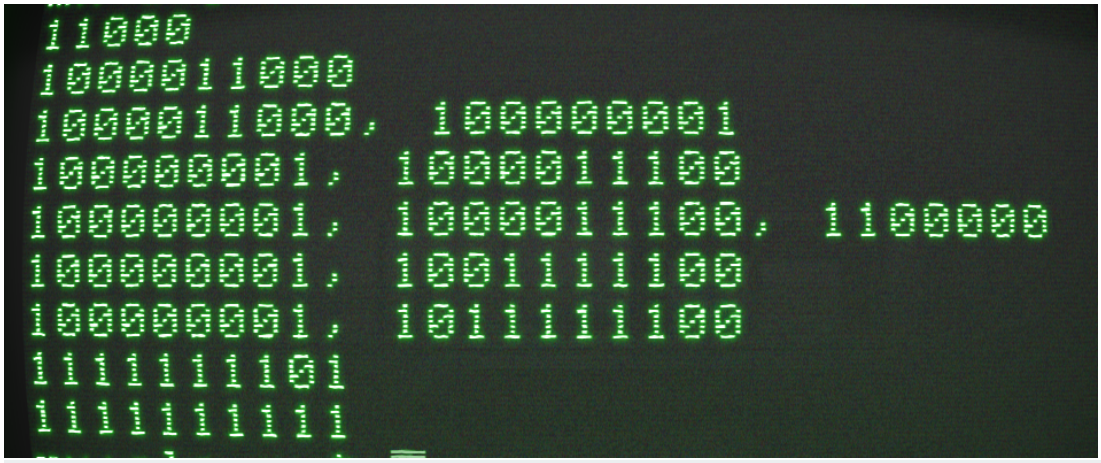
5 COMPLEXITY

It is absolutely clear that the complexity of both functions is linear with respect to the amount of these subsets, and these subsets are in worst case equal to the set in question, so the temporal complexity is $O(n)$ and the spatial complexity, is almost the same for defining the problem, only are needed bits, so the spatial complexity is $O(n)$.


UNION CONNECTED UNT ALGORITHM
TIME (s) vs SIZE
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0
2000
4000
6000
8000
10000

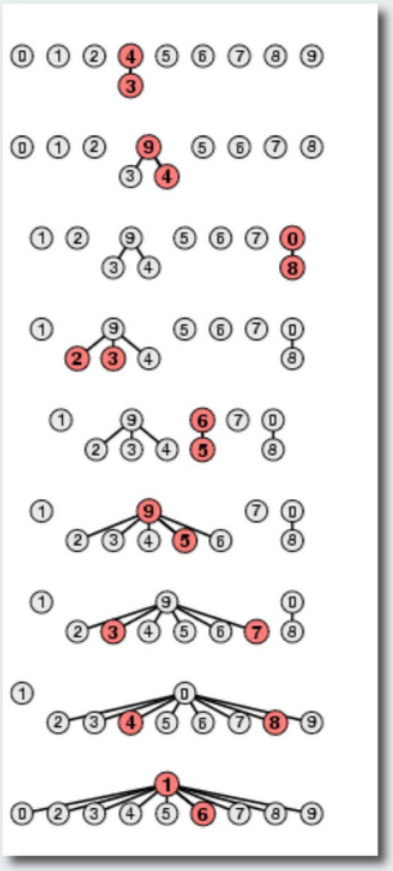
6 EXAMPLE FROM STANDARD PUBLICATIONS

```
universe = []

def union(a, b):
    element = (1 << a) + (1 << b)
    idx = 0
    while idx != len(universe):
        if universe[idx] & element:
            universe[idx] |= element
            element = universe[idx]
            del universe[idx]
            idx -= 1
        idx += 1
    universe.append(element)

def connected(a, b):
    element = (1 << a) + (1 << b)
    for item in universe:
        if item & element:
            return item
    return 0

def print_universe():
    print(', '.join([bin(element)[2:] for element in universe]))

if __name__ == '__main__':
    union(3, 4)
    print_universe()
    union(4, 9)
    print_universe()
    union(8, 0)
    print_universe()
    union(2, 3)
    print_universe()
    union(5, 6)
    print_universe()
    union(5, 9)
    print_universe()
    union(7, 3)
    print_universe()
    union(4, 8)
    print_universe()
    union(6, 1)
    print_universe()
```



3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	9	9	5	6	7	8	9
8-0	0	1	2	9	9	5	6	7	0	9
2-3	0	1	9	9	9	5	6	7	0	9
5-6	0	1	9	9	9	6	6	7	0	9
5-9	0	1	9	9	9	9	9	7	0	9
7-3	0	1	9	9	9	9	9	9	0	9
4-8	0	1	0	0	0	0	0	0	0	0
6-1	1	1	1	1	1	1	1	1	1	1

problem: many values can change



7 DISCLAMER

All content is original and of my authorship, there are no partners or affiliations, only used basic concepts of the theory of computational complexity, and standard definitions, giving references is impossible, are part of the general culture. the UNT is acronim for the UNIVERSAL NUMBER THEORY, look at my other papers.

8 LICENCE