


---

# Randomized Algorithms: Assignment 2

---

Adrià Lisa <adria.lisa@estudiantat.upc.edu>  <Repository>

For this assignment, we are asked to implement twelve variants of Union-Find, and measure their performance in terms of the number of pointer updates, and the total path length of the data structure. I used C++ for the task, the whole code is available on my Github repository.

## 1 The Data Structure

I designed a flexible data structure, where the functions for "Union" and "Find" are made explicit upon construction. The constructor `UnionFind(...)` takes as arguments the initial number of sets and two enum types used to specify the strategy for unions and path compression.

A private vector `V` is the core of the structure. If `i` is not a root, the value of `V[i]` will be the parent of `i`.

I found convenient to keep as the name of the data structure a string with the chosen combination of union and find strategies ("`QU_NC`", "`UR_PH`", "`UW_FC`", ...).

It was also worth it to keep track of the number of disjoint sets in the private field `int nblocks`, to ease the computation of the desired measures on the data structure.

```
using Vec = std::vector<int>;
```

```
class UnionFind {
private:
    Vec V;
    FindFunction findFunc;
    UnionFunction unionFunc;
    int n_blocks;

public:
    int Find(int x);
```

```
void Union(int x, int y);
```

```
UnionFind(
    int n,
    UnionMethod u,
    FindMethod f
);
```

```
string name;
```

```
int PathLength(int x);
int TotalPathLength();
int TotalNumberOfChildren();
bool AllJoined();
};
```

Most of the functions on the data structure serve the experimentation process, but do note that only the constructor, `int Find` and `void Union` are strictly necessary.

## 2 Union Methods

These are the listings of my implementations of the three possible choices for the "Union" strategy. Keep in mind that for quick union, root nodes point to themselves (`V[r] = r`), whereas in the other two strategies root nodes point to a negative number (`V[r] = -measure`), corresponding to the size/weight of their sets, or the rank in the case of union-by-rank.

All the methods fit in the type `UnionFunction` defined as:

```
function<void(std::vector<int>&, int, int)>
```

- Unweighted quick-union ("`QU`")

```
quick_union(Vec& root, int rx, int ry) {
    root[ry] = rx;
}
```

- Union-by-weight ("`UW`")

```

union_by_weight(Vec& P, int rx, int ry) {
    if (P[rx] >= P[ry]) {
        // rx is the shortest
        P[ry] += P[rx];
        P[rx] = ry;
    } else {
        P[rx] += P[ry];
        P[ry] = rx;
    }
}

```

- Union-by-rank ("UR")

```

union_by_rank(Vec& P, int rx, int ry) {
    if (P[rx] >= P[ry]) {
        // rx is the shortest
        P[ry] = std::min(P[ry], P[rx]-1);
        P[rx] = ry;
    } else {
        P[rx] = std::min(P[rx], P[ry]-1);
        P[ry] = rx;
    }
}

```

Do note that this methods work directly from the roots, thus we always have to precede it with calls to the Find method. The actual implementation of the Union method in the class is as follows:

```

void UnionFind::Union(int x, int y) {
    int rx = Find(x);
    int ry = Find(y);
    if (rx != ry) {
        unionFunc(V, rx, ry);
        --n_blocks;
    }
}

```

### 3 Find Methods

All the "Find" methods, independently of the path compression, require a condition to check whether a node is a root. For quick-union that is  $V[i] == i$ , whereas for the other union strategies it is  $V[i] < 0$ .

In the following, we list the implementation of four path compression strategies for

union-by-weight and union-by-rank. There is a different version of each for quick-union, and it only changes the condition for checking if a node is a root.

- No path compression "NC"

```

int no_compression_find (Vec& P, int x) {
    while (P[x] >= 0) x = P[x];
    return x;
}

```

- Full path compression "FC"

```

int full_compression_find (Vec& P, int x) {
    if (P[x] < 0) return x;

    P[x] = full_compression_find(P, P[x]);
    return P[x];
}

```

- Path-splitting "PS"

```

int path_splitting_find (Vec& P, int x) {
    while (P[x] >= 0){
        if (P[P[x]] < 0){
            return P[x];
        }
        int aux = P[x];
        P[x] = P[P[x]];
        x = aux;
    }
    return x;
}

```

- Path-halving "PH"

```

int path_halving_find (Vec& P, int x) {
    while (P[x] >= 0){
        if (P[P[x]] < 0){
            return P[x];
        }
        P[x] = P[P[x]];
        x = P[x];
    }
    return x;
}

```

## 4 Experimental setup

A common sequence of pairs of integers for all the 12 combinations of "UnionFind" is generated with a random seed, using the method `shuffle(...)` from the `<random>` c++ header.

```
generateShuffledPairs(int n) {
    vector<pair<int, int>> pairs;
    for (int i = 0; i < n; ++i) {
        for (int j=i+1; j<n; ++j) {
            pairs.push_back({i, j});
        }
    }
    // random number generator rd
    random_device rd;
    // Mersenne Twister seeded with rd
    mt19937 g(rd());
    shuffle(pairs.begin(), pairs.end(), g);
    return pairs;
}
```

After initializing every "UnionFind", we perform unions of the pairs in the sequence, until all are joined into a single disjoint set. Every `del` ( $= \Delta$ ) unions, we take measures of the Total Path Length (TPL) and the Total number of Pointer Updates (TPU).

The methods for computing TPL are implemented inside the data structure, and consist on a basic walk from all the nodes.

```
int UnionFind::PathLength(int x){
    int distance_to_representative = 0;
    while (V[x] >= 0 && V[x] != x)
    {
        x = V[x];
        distance_to_representative++;
    }
    return distance_to_representative;
}

int UnionFind::TotalPathLength(){
    int n = V.size();
    int total_length = 0;
    for (int x = 0; x < n; ++x)
        total_length += PathLength(x);
    return total_length;
}
```

The measure TPU is dependent on the path compression strategy. For "NC" it is always zero, and we have a smart way to compute it for "FC" and "PH":

$$TPU = TPL - \sum_{r \text{ is root}} \#\{\text{children of } r\}.$$

However, that formula is not correct for path-halving. In this setup, the amount of pointer updates required on a Find operation on the node of depth or path-length  $k$  is given by  $\lfloor k/2 \rfloor$ . Therefore,

$$TPU = \sum_{i \text{ node}} \left\lfloor \frac{PL(i)}{2} \right\rfloor$$

```
int TPU(UnionFind& uf, FindMethod f){
    if (f == FindMethod::NO_COMPRESSION)
        return 0;
    if (f == FindMethod::PATH_HALVING){
        int TPU = 0;
        for (int i = 0; i < n; i++){
            TPU += uf.PathLength(i)/2;
        }
        return TPU;
    }
    // if it is FC or PS
    return TPL(uf) - uf.TotalNumberOfChildren();
}
```

We save those two quantities divided by the current amount of unions in a .csv file. The main method of our experimental setup contains the following code snippet:

```
// u::UnionMethod, f::FindMethod
UnionFind UF(n, u, f);
TPL_file << UF.name;
TPU_file << UF.name;
for (int i = 0; i < m; i++) {
    if (i % del == 0 and i > 5) {
        TPL_file << "," << (float)TPL(UF)/(float)i;
        TPU_file << "," << (float)TPU(UF, f)/(float)i;
    }
    if (UF.AllJoined()) break;
    UF.Union(pairs[i].first, pairs[i].second);
}
```

The full code for the experimental setup can be found in the file "experiments.cpp" on the already mentioned Github repository.

## 5 Results

The figures on the results obtained were made using Python's library `matplotlib`. The scripts for the plotting hold no particular interest. To save time, they were put together through a conversation with ChatGPT 3.5 (no shame on that, these days).

We can compare the results for different path compression heuristics on the Quick Union strategy on figures 1, 2, 3. Each uses the data for different values on  $n$ , the number of elements in the data structure. The results for the no-compression strategy ("**NC**") quickly outgrow the others in the TPL metric. For that reason, that particular plot is visualized in logarithmic scale.

Similarly, figures 4, 5, 6 showcase the results using union-by-rank, and figures 8, 9, 10 show the results for the union-by-weight. We can see that the results in this figures are all very similar, with a clear difference for the no-compression "Find" strategy.

To give a fair comparison of each strategy, we define the overall cost depending on the "Find" method  $f$  as:

$$\begin{aligned} TPL & \text{ if } f = \text{"NC"} \\ 2TPL + \epsilon TPU & \text{ if } f = \text{"FC"} \\ TPL + \epsilon TPU & \text{ if } f = \text{"PS"} \\ TPL + \epsilon TPU & \text{ if } f = \text{"PH"} \end{aligned}$$

Where the  $\epsilon$  parameter represents the ratio between the computational cost between accessing a pointer and updating it.

For this comparison, I took as common value  $n = 5000$ . The results for the Quick-Union were far worse than all the others, so I omitted them from this last plots for the sake of clarity.

The values for  $\epsilon$  that were considered are  $\epsilon = 0.1, 1, 2, 10, 100$ , and their respective plots are figures 11, 12, 13, 14, 15. The values  $\epsilon = 0.1, 100$  seem a bit far fetched, but still are a nice way to visualize the extreme cases.

Quite surprisingly, we can see that  $\forall \epsilon$  "**FC**" has a higher cost than "**NC**", and that "**PS**" and "**PH**" have virtually the same cost. Moreover, the matter of "**NC**" versus "**PH**" or "**PS**" depends on the cost of updating the pointer ( $\epsilon$ ), but mostly on the expected number of Union operations. If we perform a small amount of unions, the paths will not be so large, so it will be more sensible to not bother on compressing them. But for large amounts of unions, the paths will be large, and the figures show that it will be worth it to compress them.

## 6 Conclusion

My personal conclusion is that the best combination is "**UWPS**", because all the combinations "**UW**" / "**UR**" + "**PS**" / "**PH**" have the best performance when we expect a large amount of operations (which is when we care about performance in the first place!), and I feel that "**UWPS**" is easier to understand, analyze and implement.

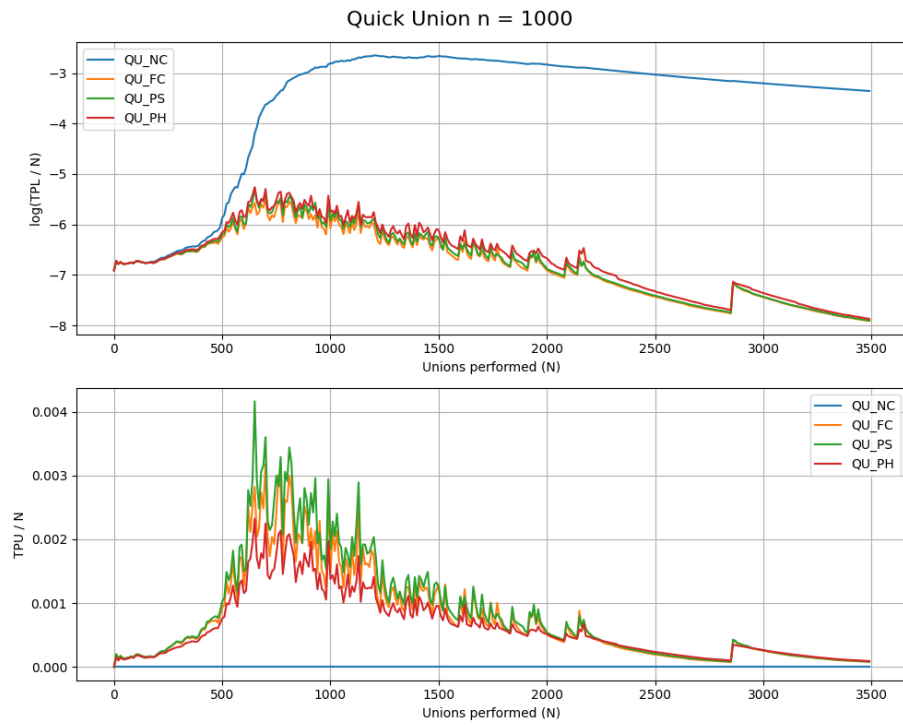


Figure 1

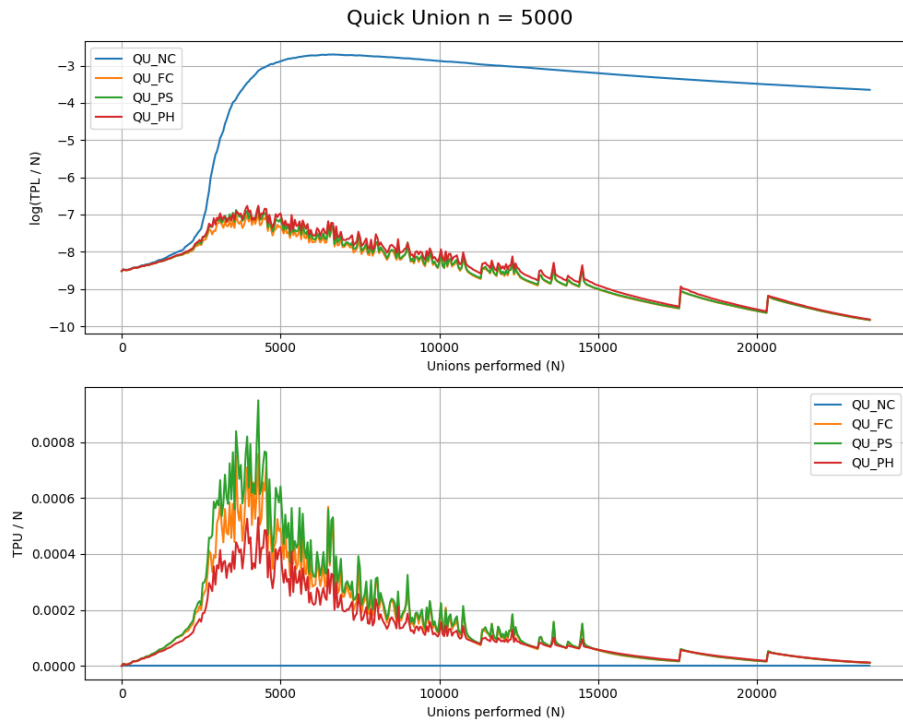


Figure 2

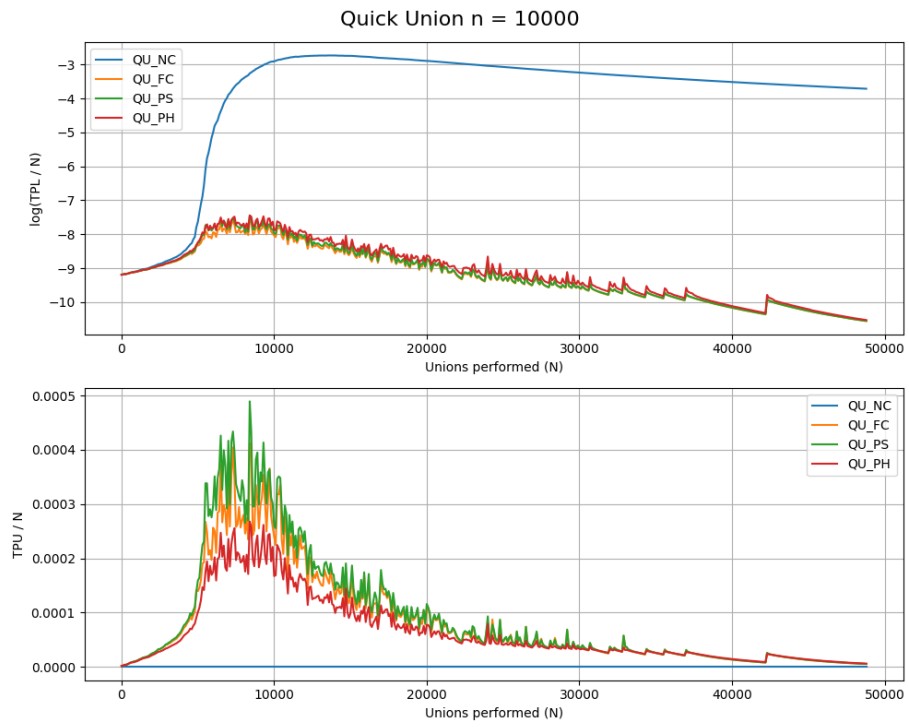


Figure 3

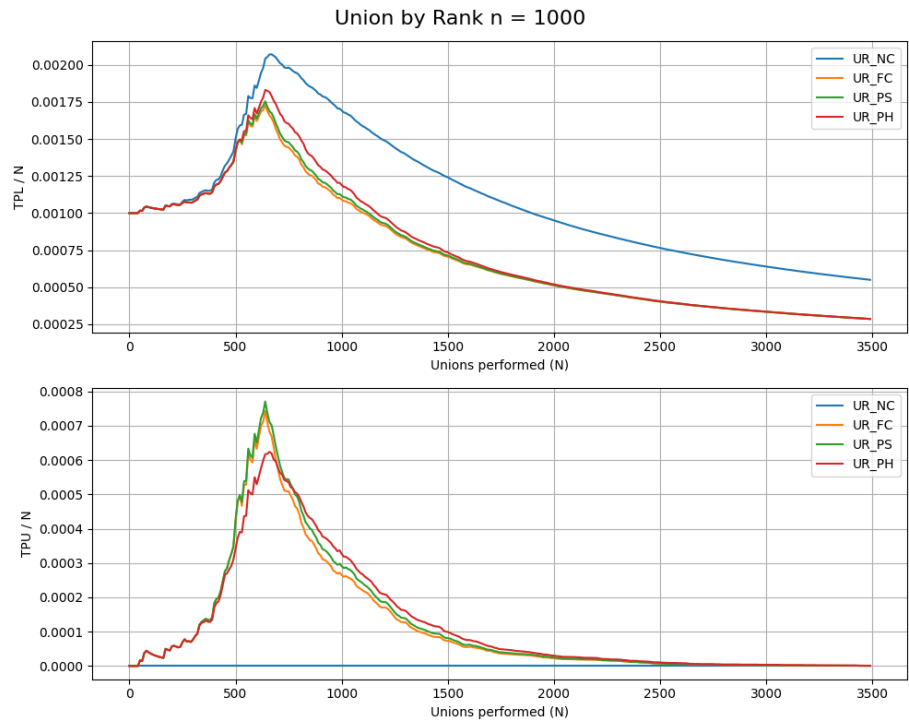


Figure 4

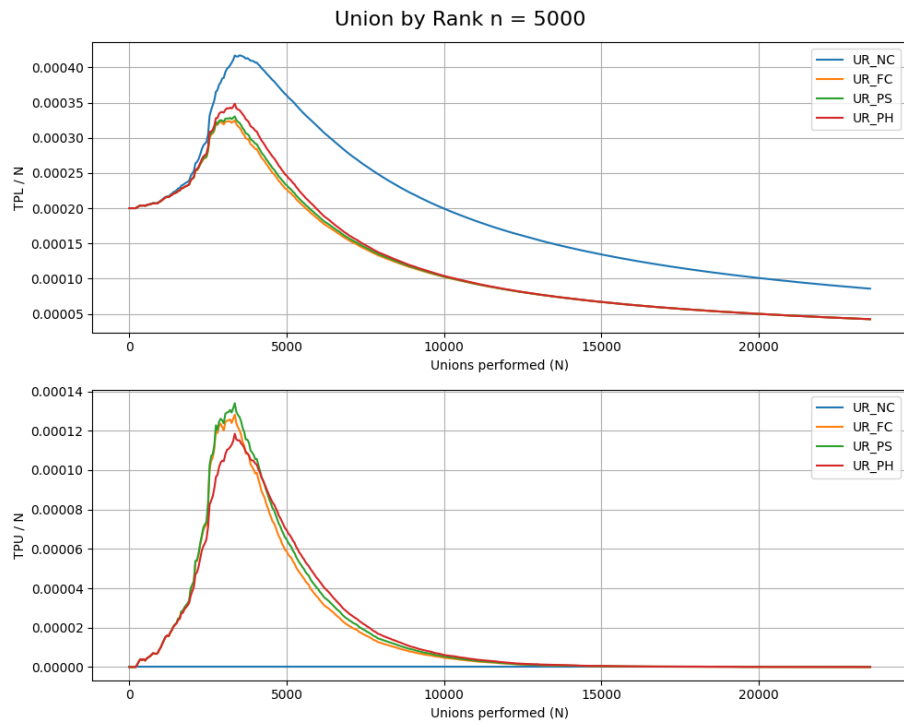


Figure 5

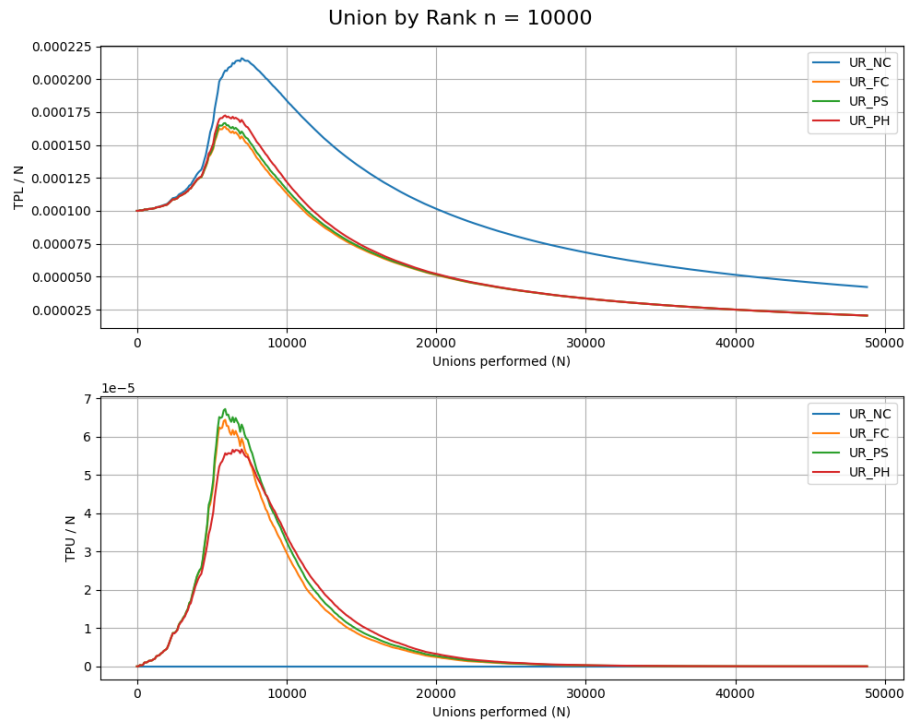


Figure 6

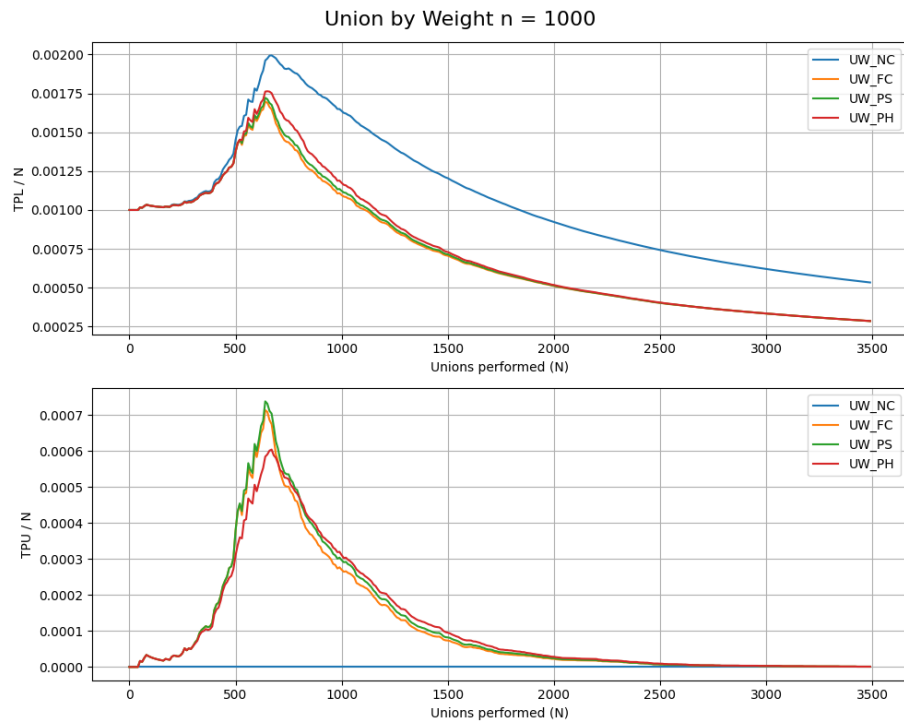


Figure 7

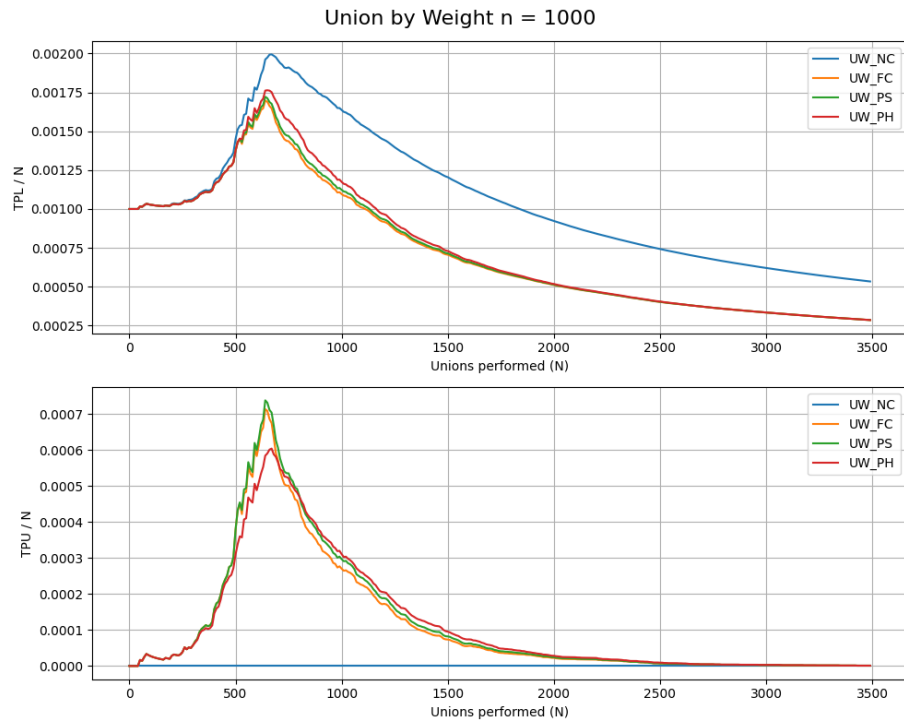


Figure 8



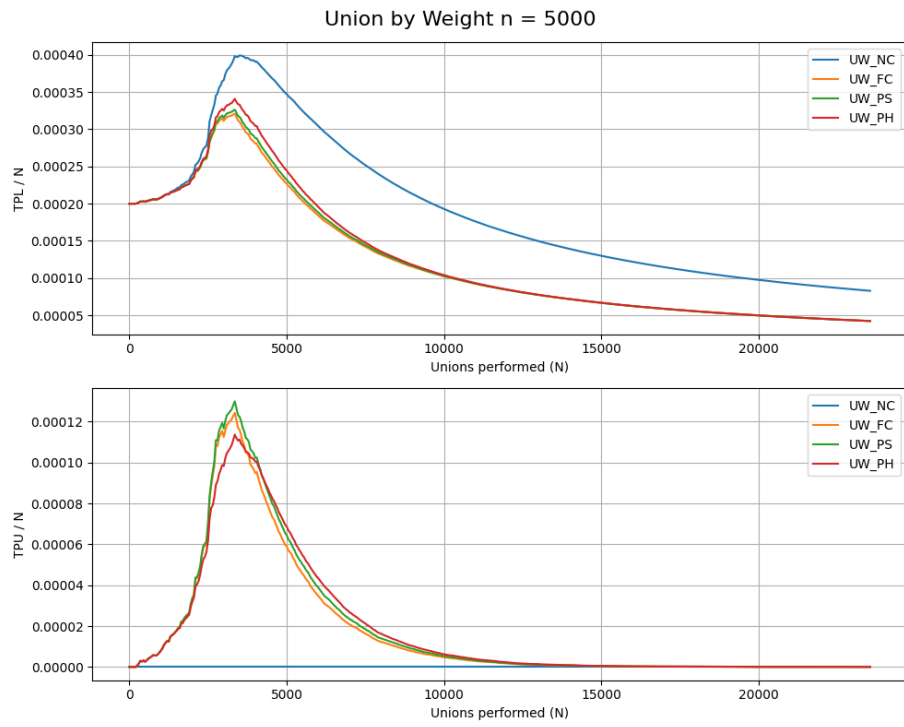


Figure 9

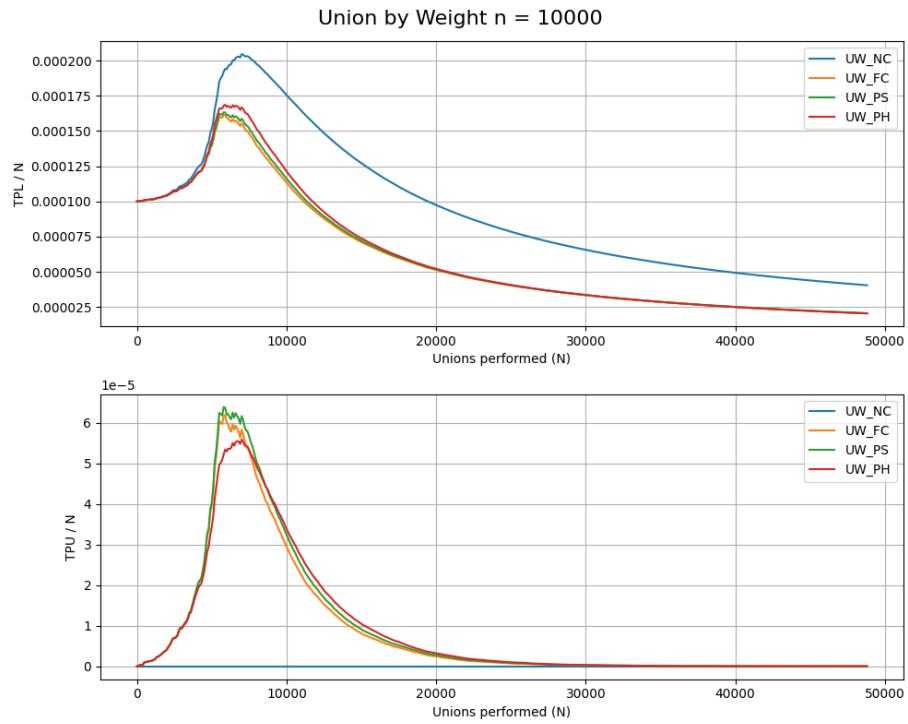


Figure 10

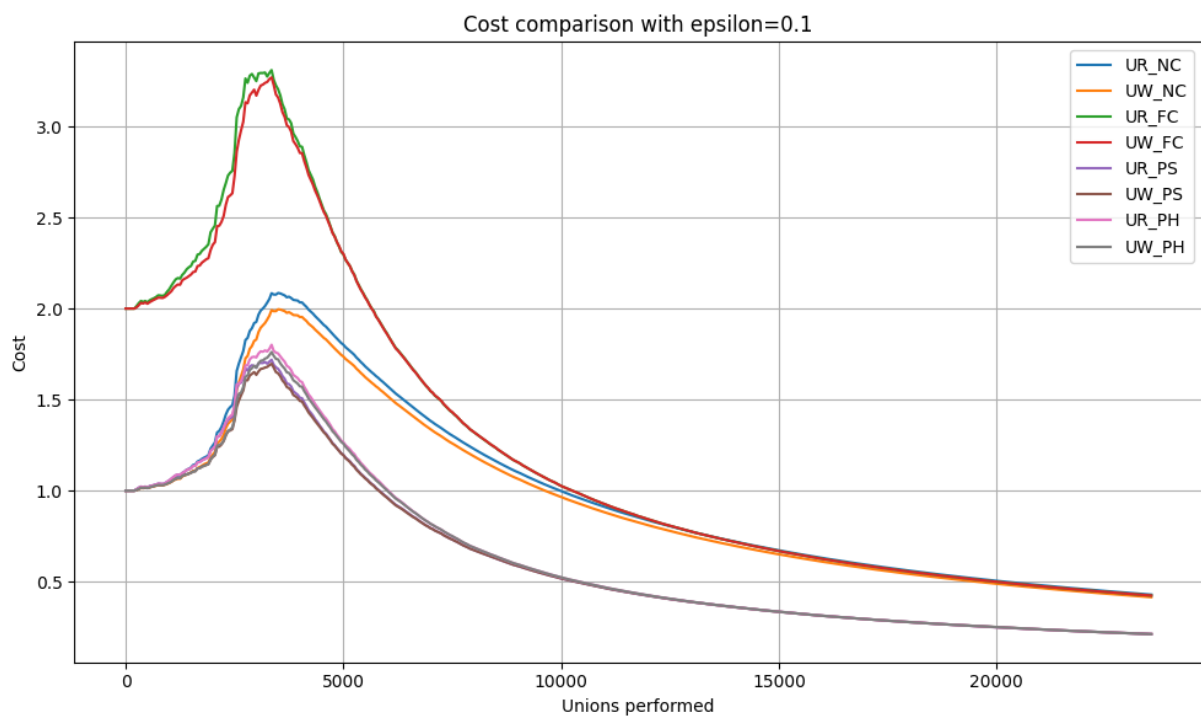


Figure 11

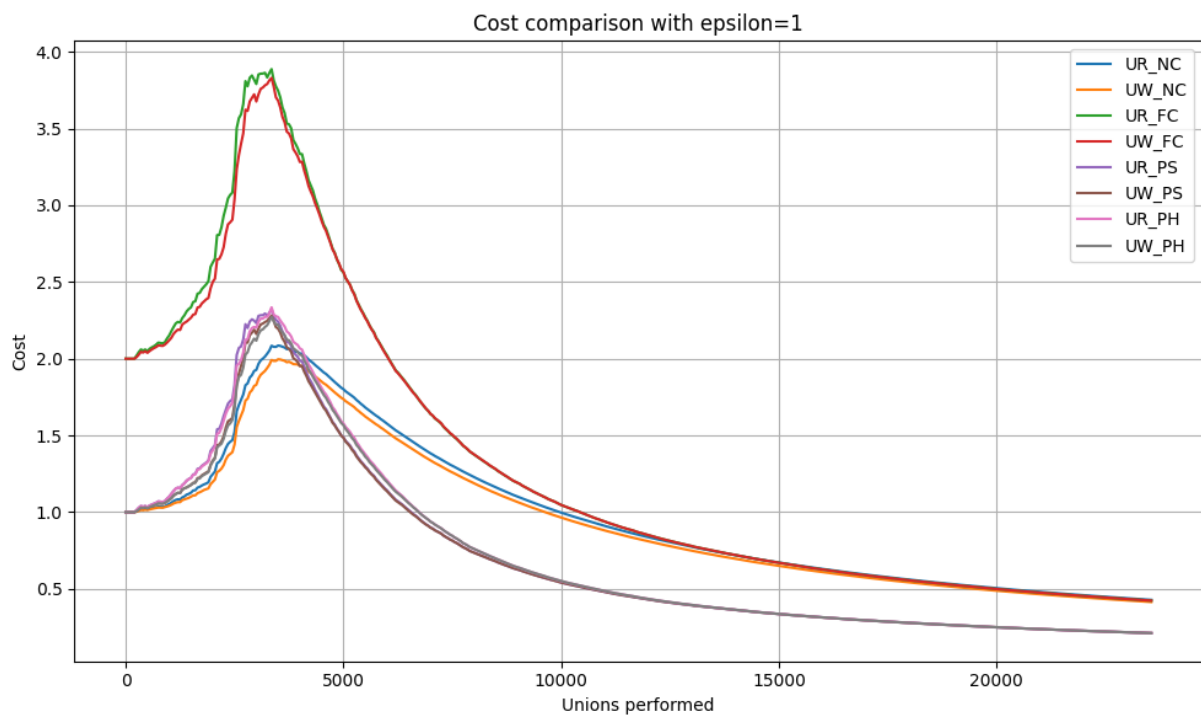


Figure 12

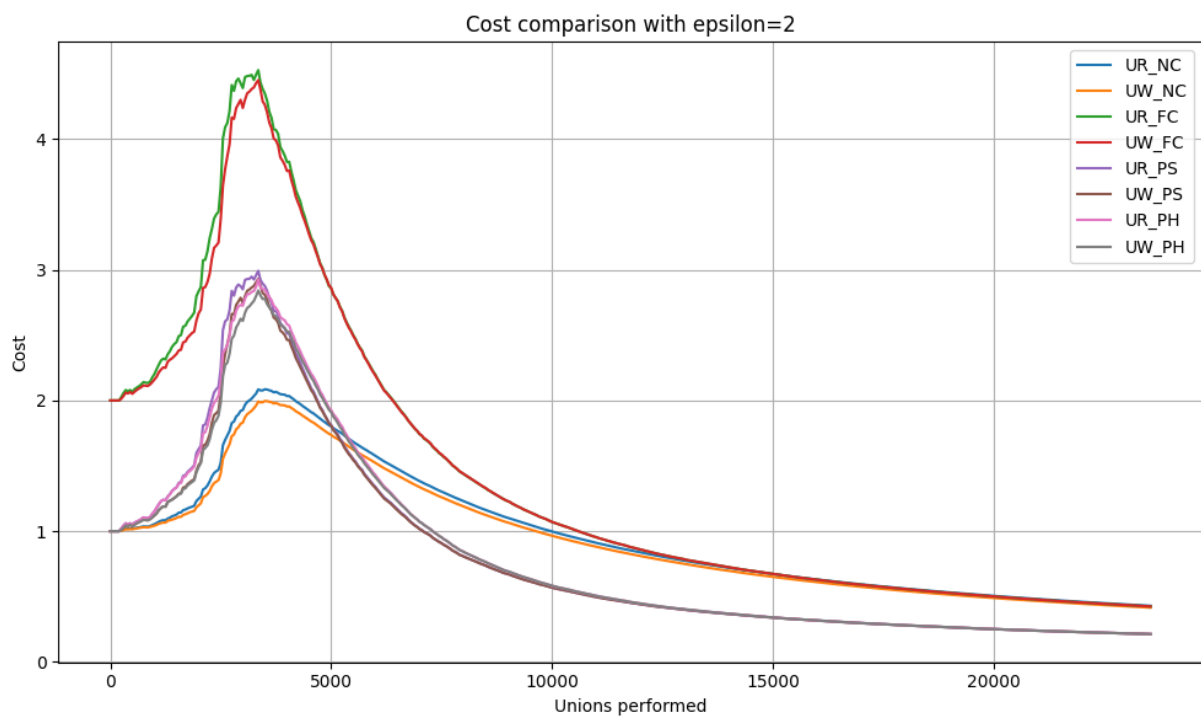


Figure 13

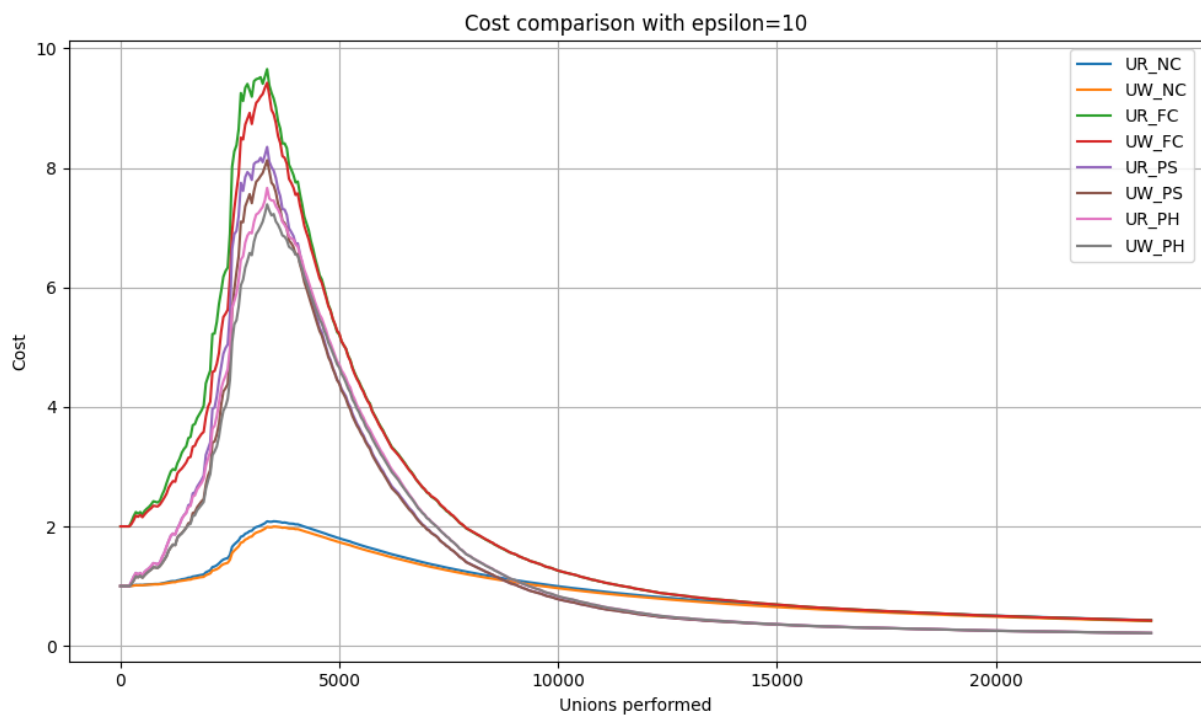


Figure 14

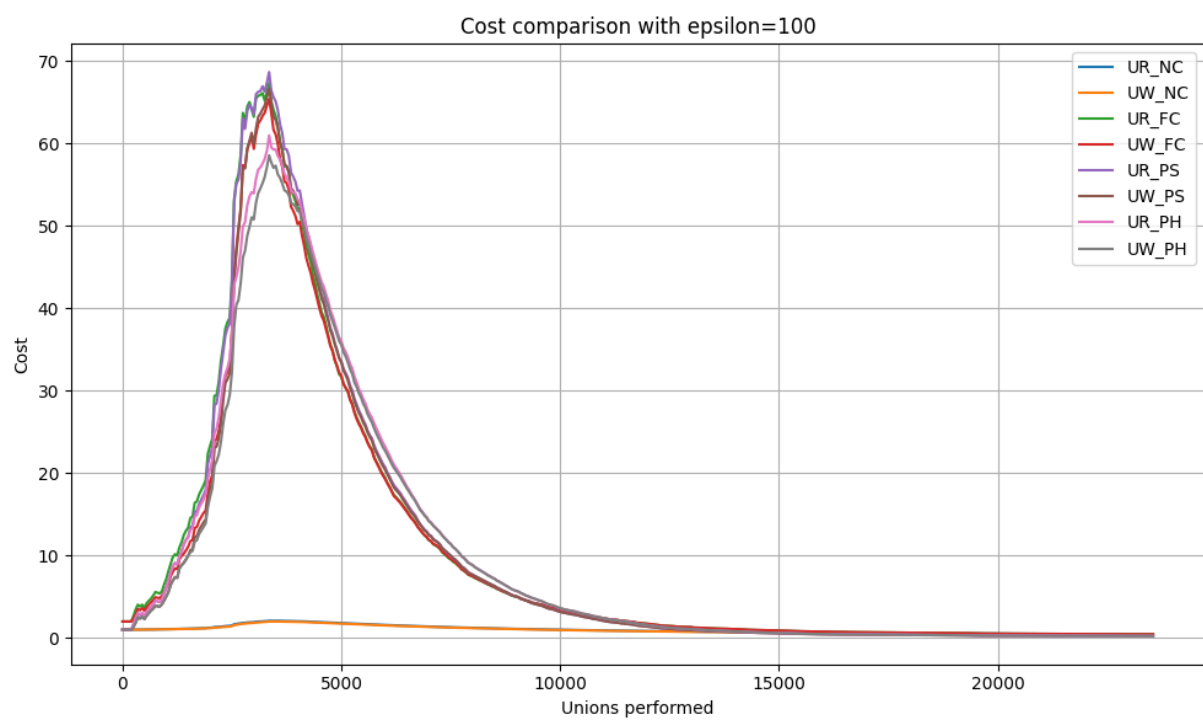


Figure 15