4 - Empirical Study of Union-Find

Jakob Eberhardt
jakob.eberhardt@estudiantat.upc.edu

May 7, 2025

Contents

Listings

List of Figures

1 Introduction

In this assignment, we study different heuristic strategies for finding and merging elements in disjoint sets. To this end, we set up an experiment to benchmark the respective implementations on synthetic data. To achieve machine-independent results we measure the following metrics:

- Total Path Length (**TPL**): The distance to its representative of every element in the data structure summed up. This measures to which degree the representation of our sets degenerated to a list-like data structure.
- Total Pointer Updates (**TPU**): Measures the number of required pointer updates during a `Find` operation for a given element.
- Total Cost: A heuristic-specific linear combination of TPL and TPU.

In section 2, we introduce the project structure and the policy-independent implementation. Section 3 includes the straight-forward `QuickUnion` (**QU**) and `No Compression` (**NC**) implementation and the degeneration problems that go along with it. The heuristics sections 4 and 5 include the different policies that can be applied for coping with these problems. In sections 6 and 7, we discuss the experimental setup and the results of the practical experiment.

2 Basic Implementation

A `Union Find` data structure allows us to store disjoint sets in an efficient manner. In the following section, we introduce the project's common setup for the experiment. The data structure must feature three operations, adding a new set, merging two sets, and finding the representative member of a set. Hence, we can define a virtual class as an interface for our actual implementation, as can be seen in listing 1. In addition, we add the respective functions to access static properties and run time metrics of our data structure.

```
class DisjointSet {
public:
    virtual ~DisjointSet() = default;

    virtual void makeSet(int x)                    = 0;
    virtual int find(int x)                        = 0;
    virtual void unionSets(int x,int y)            = 0;
    virtual int getParent(int x)             const = 0;
    virtual int countSets()                  const = 0;
    virtual int depth(int x)                 const = 0;
    virtual long pointerUpdatesDuringFind(int x) const = 0;
}
```

Listing 1: The virtual class in `DisjointSet.h` defines the basic interface for our implementation and benchmarking setup

This interface is implemented by the `UnionFind` class as can be seen in listing 2. It is parameterized with two templates, a Union policy (`U`) which determines how two roots are linked, and a Find policy (`F`) which defines how we walk and compress the path to a root. This code design will allow us to easily compile and run every combination of $U \times F$ later on while avoiding redundant code. The data fields of our structure comprise the three vectors. For **QU**+**NC**, the only relevant data is stored in `parent_` vector. It keeps the index of the parent element for every element. The `size_` and `rank_` vectors will be used for union heuristics.

```
template<class U, class F>
class UnionFind : public DisjointSet
...
private:
    std::vector<int> parent_, size_, rank_;
```

Listing 2: Policy-based Implementation of the interface in `UnionFind.h`

The data can be initialized using the constructor which will allocate three parallel arrays of size `n`. It initializes every vertex as an isolated root and sets their `size` and `rank` to one and zero respectively. The `makeSet` function is for testing and ensures proper initialization and resizing if necessary.

```
public:
    explicit UnionFind(int n)
        : parent_(n), size_(n,1), rank_(n,0)
    { for (int i=0;i<n;++i) parent_[i]=i; }

    void makeSet(int x) override {
        if (x >= (int)parent_.size()) {
            int old = parent_.size();
            parent_.resize(x+1);
            size_.resize(x+1,1);
            rank_.resize(x+1,0);
            for (int i=old;i<=x;++i) parent_[i]=i;
        } else {
            parent_[x]=x; size_[x]=1; rank_[x]=0;
        }
    }
```

Listing 3: Constructor and data initialization in `UnionFind.h`

In listing 4, we can see how the `UnionFind` class implements common parts of the `unionSets` and `find` functions before the respective policy-specific code is in-lined. To minimize complexity, we pass all the parameters, independent of the heuristic which is later used, e.g. the dummy variable `upd` which could be used to track pointer updates.

```
    int find(int x) override {
        long upd=0;
        return F::find(x,parent_,upd);
    }

    void unionSets(int a,int b) override {
        int r1=find(a), r2=find(b);
        if (r1==r2) return;
        U::unite(r1,r2,parent_,size_,rank_);
    }
```

Listing 4: The policy-independent parts of the `unionSets` and `find` functions in `UnionFind.h`. The parameters are passed to a specific combination of heuristics, e.g. QuickUnion without compression.

3 QuickUnion without Compression & Problem Statement

If we do not employ any compression strategy while finding the set representative, the `find` function simply chases the pointer until we encounter the set representative. Hence, we never update and pointer and we only read every node once, `followMult` is therefore one.

```
struct NC {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (x != p[x]) x = p[x];
        return x;
    }
    static long updatesGivenDepth(int depth) { return 0; }
    static double followMult() { return 1.0; }
};
```

Listing 5: The `find` implementation without any additional compression considerations in `FindPolicies.h`

As mentioned, upon initialization, each element belongs to its own set, meaning its parent value points to its own index (`parent_[i] == i`). Therefore, we always start with a data structure that looks like figure 1.

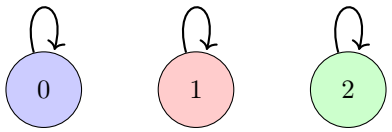

Figure 1: Initial configuration: three separate nodes (0–2). Each node points to itself (self-loop), indicating that each is initially its own set.

The policies for how two representatives are linked upon a union operation are defined in the `UnionPolicies.h` file. For many use cases, it is essential how roots are attached in order to keep the resulting tree balanced. For QuickUnion, however, we always assign the second operand as the parent of the first, as can be seen in listing 6.

```
struct QU {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    { parent[r1] = r2; }
};
```

Listing 6: The QuickUnion policy defined in `UnionPolicies.h`

Following the example from figure 1, if we apply `union(0,1)` seen in figure 2 and `union(1,2)` in figure 3, we can see how a sequence of operations like this can cause an unbalance data structure or even a degeneration to a list.

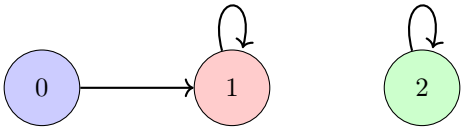

Figure 2: After `union(0,1)`: Element zeros's parent becomes element one, forming a chain fragment (0 → 1). Element one remains the representative of the set containing nodes zero and one, while element two is still a separate set.

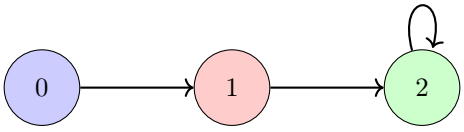

Figure 3: After `union(1,2)`: Element one's parent becomes element two, resulting in the complete chain: 0 → 1 → 2. Element two is now the representative of the set.

In the following, we will discuss and compare heuristics for union and compression techniques during find operations to prevent this behavior.

4 Heuristics for `Union`

Every union operation will cause two find operations. The time required for each find operation is proportional to the amount of parent pointers we have to follow. Hence, we aim to keep the trees that represent our sets as flat as possible, independent of the input order. Additionally, we want to keep the amount of pointers that have to be updated to flatten a tree as low as possible.

4.1 Union by Weight

For the Union by weight heuristic, we use the `size` array to track how many elements are represented by the given root. We try to keep the trees shallow by putting the smaller tree under the larger one by comparing their size S_i.

```
struct UW {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    {
        if (size[r1] < size[r2]) std::swap(r1,r2);
        parent[r2] = r1;
        size[r1] += size[r2];
    }
};
```

Listing 7: The Union by weight policy defined in `UnionPolicies.h`

The height of the tree which remains a root may only grow by a maximum factor of two since

$$S' = S_1 + S_2 \leq S_1 + S_1 = 2 \cdot S_1$$

For example, for the sequence `union(0,1)` $\rightarrow$ `union(1,2)`, which previously resulted in a height of $n-1$ depicted in figure 2 and 3, we now maintain a height of one, as can be seen in figure 4 and 5.

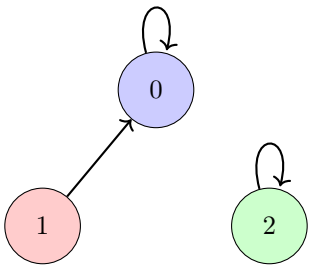

Figure 4: After `union(0,1)` with the union by weight policy. If the two subtrees weigh the same, the right operand becomes the root or respectively the representative of the set. Element 2 is still alone.

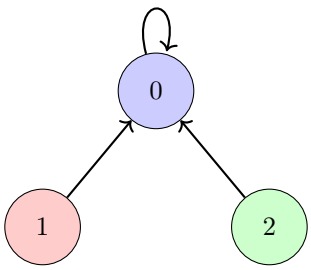

Figure 5: After `union(1,2)` with Union by weight: the smaller subtree (root 2) is attached to the larger one (root 0).

In the worst case, where we always have to merge equal-size trees like in this sequence: `union(0,1)` → `union(2,3)` → `union(4,5)` → `union(6,7)` → `union(0,2)` → `union(4,6)` → `union(0,4)` we get a tree with the maximum height of $\Theta(\log n)$

4.2 Union by Rank

In Union by rank the rank of a root starts at zero and is monotone, meaning we never decrease it, e.g. during path compression. Each increment happens precisely when two equal–ranked trees merge, which at least doubles the size of the resulting tree, so after k increments the tree contains $\geq 2^k$ elements. This ties the rank to the maximum possible size of the set which is n. Hence for a set of size n we have $rank \leq \lfloor \log_2 n \rfloor$ and, because the actual height is never larger than the rank,

$$height \leq rank \ \leq \lfloor \log_2 n \rfloor$$

which gives us the same $\Theta(\log n)$ height guarantee as union by size while requiring less bookkeeping, because we just have to occasionally update a rank.

```
struct UR {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    {
        if (rank[r1] < rank[r2]) std::swap(r1,r2);
        parent[r2] = r1;
        if (rank[r1] == rank[r2]) ++rank[r1];
    }
};
```

Listing 8: The Union by rank policy defined in `UnionPolicies.h`

5 Compression Heuristics for `Find`

We can employ different pointer compression heuristics during find operations to guarantee certain properties after a call. Depending on the use case or the targeted hardware, we can modify the forest more or less aggressively, as the following examples will showcase.

5.1 Full Compression

Full compression is the most aggressive heuristic. We follow the parent pointers to the root, then we make every node on that path point directly to the representative. Hence, the former tree structure is flattened to a star at the cost of a second traversal to update the pointers.

```
struct FC {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        if (p[x] != x) {
            p[x] = find(p[x], p, upd);
            ++upd;
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return depth; }
    static double followMult() { return 2.0; }
};
```

Listing 9: The Full Compression policy defined in `FindPolicies.h`

5.2 Path Splitting

In path splitting, we split one long edge into two shorter edges. Namely, we skip over $\langle x, \text{parent}(x)\rangle$ and add $\langle x, \text{grandparent}(x)\rangle$ along with $\langle \text{parent}(x), \text{grandparent}(x)\rangle$ upon a single top-to-root pass. Hence, we get a height ≤ 2 on the visited path.

```
struct PS {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (p[x] != p[p[x]]) {
            int parent = p[x];
            p[x] = p[parent];
            ++upd;
            x = parent;
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return depth/2; }
    static double followMult() { return 1.0; }
};
```

Listing 10: The Path Splitting policy defined in `FindPolicies.h`

5.3 Path Halving

In path halving we shortcut every second edge encountered on the top-to-root scan by making each visited node point directly to its grandparent, e.g., $\langle x, \text{parent}(x)\rangle$ becomes $\langle x, \text{grandparent}(x)\rangle$. Because only alternate vertices are rewired, the depth of every node on the walked path is reduced from d to $\lceil d/2 \rceil$, *halving* the distance to the root. The whole path is processed in a single pass, so we obtain this depth reduction with exactly one traversal and about half as many pointer writes as full path compression.

```
struct PH {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (p[x] != p[p[x]]) {
            p[x] = p[p[x]];
            ++upd;
            x = p[x];
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return (depth+1)/2; }
    static double followMult() { return 1.0; }
};
```

Listing 11: The Path Halving policy defined in `FindPolicies.h`

6 Experimental Setup

During experimentation, we use three instance sizes, small ($n = 1000$), medium ($n = 5000$) and large ($n = 10000$). The experimental setup is defined in the `main.cpp` file. The function `measureMetrics` scans all n vertices after each sampling point so that the recorded totals TPL and TPU reflect the entire forest at that moment for measurements at every Δ unions.

```
struct Metrics { long tpl = 0; long tpu = 0; };

Metrics measureMetrics(const DisjointSet& uf, int n)
{
    Metrics m;
    for (int v = 0; v < n; ++v) {
        m.tpl += uf.depth(v);
        m.tpu += uf.pointerUpdatesDuringFind(v);
    }
    return m;
}
```

Listing 12: The `measureMetrics` in `main.cpp`

The helper function `indexToPair` converts a rank $k \in 0, \ldots, \binom{n}{2} - 1$ into the unique unordered pair (i, j), turning the set of all pairs into a contiguous integer range that can be shuffled cheaply and without storing explicit pairs.

```
inline std::pair<int,int> indexToPair(long long k)
{
    long long i = static_cast<long long>( (1 + std::sqrt(1 + 8.0*k)) / 2 );
    while (i*(i-1)/2 > k) --i;
    long long j = k - (i*(i-1))/2;
    return {static_cast<int>(i), static_cast<int>(j)};
}
```

Listing 13: The `indexToPair` mapping function in `main.cpp`

The class `PairPermutation` builds a vector of those ranks, shuffles it with `std::shuffle` and a fixed seed to obtain a reproducible uniform random permutation, and then delivers one fresh rank per call to `next()`, guaranteeing that no pair is ever processed twice.

```
class PairPermutation {
    public:
        PairPermutation(long long m, unsigned seed)
            : perm_(m), idx_(0)
        {
            for (long long i = 0; i < m; ++i) perm_[i] = i;
            std::shuffle(perm_.begin(), perm_.end(), std::mt19937_64(seed));
        }

        bool next(long long &k)
        {
            if (idx_ >= perm_.size()) return false;
            k = perm_[idx_++];
            return true;
        }
    private:
        std::vector<long long> perm_;
        size_t                 idx_;
    };
```

Listing 14: The `PairPermutation` Class in `main.cpp`

7 Results

From figure 6 we can see that Union-by-Rank and Union-by-Weight in combination with a compression heuristic yield the best results. QuickUnion without any compression performs the worst, as can be seen in figure 7. This showcases the importance of implementing a union heuristic beyond plain root attachment. Without any further compression, we can save orders of magnitude in terms of the expected total path length in the forest and hence a massive saving in cost, as can be seen in figure 14. For QuickUnion, we can also observe how increasing the instance size influences the shape of the TPL and Cost curves, as they grow faster as the instance size grows. Meanwhile, Union-by-weight and Union-by-rank maintain the same shape, indifferent of the input size. A surprising result of the experiment is the sharp drop in TPL and Cost when any kind of compression heuristic is employed, but most significantly when using Full Compression. As the amount of distinct blocks decreases, we likely traverse the same sets over and over again, which results in aggressive compression. The same behavior can be seen in the required pointer updates in figure 12, as many elements are already attached to the ultimate root, meaning they will never require an update again during the benchmark.

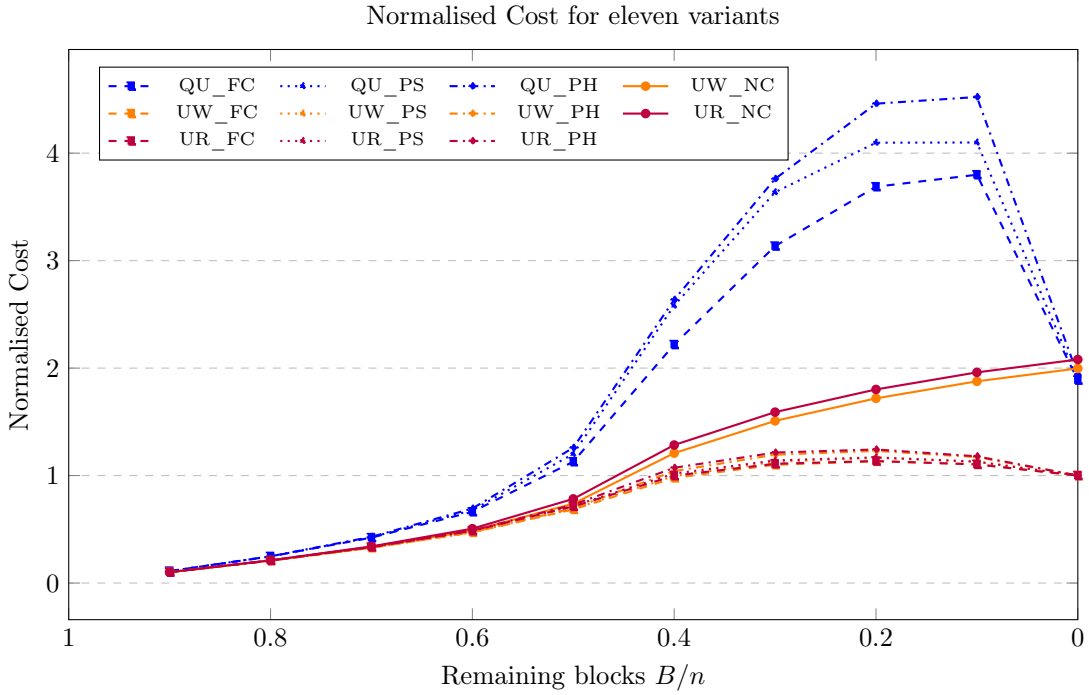

Figure 6: Normalised cost as a function of the fraction of blocks remaining (B/n) for every combination of union strategy (colour) and path-compression variant (marker / line style).

7.1 Normalised Total Path Length

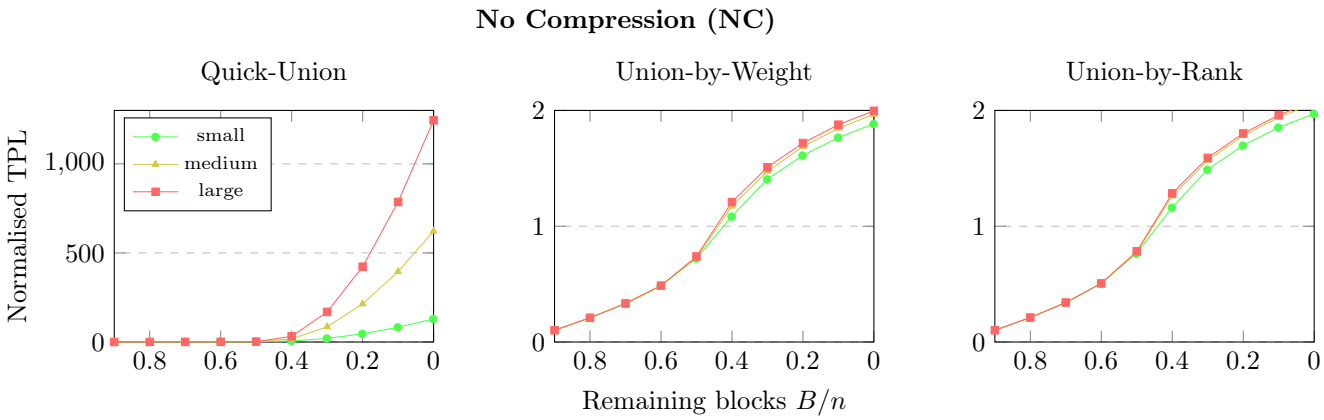

Figure 7: Normalised Total Path Length without compression.

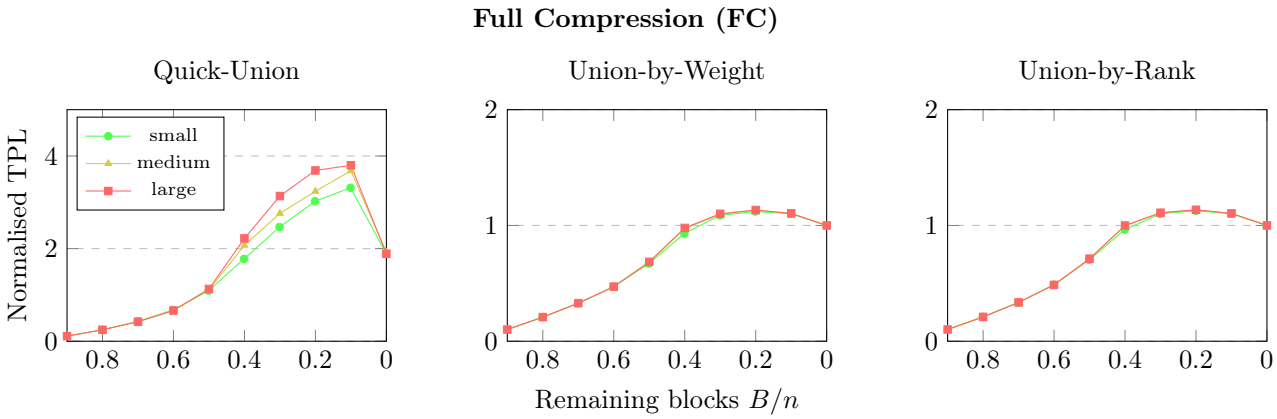

Figure 8: Normalised Total Path Length of the Full Compression strategy.

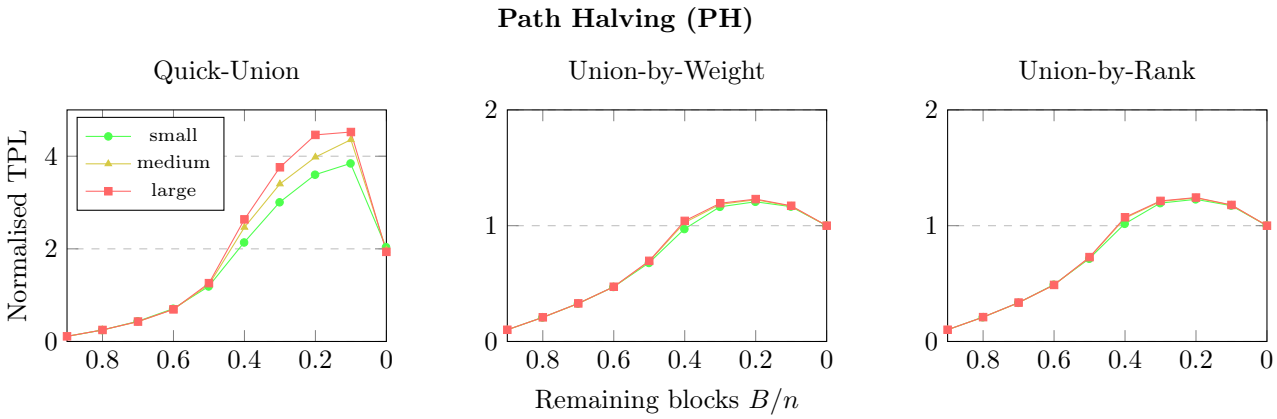

Figure 9: Normalised Total Path Length of the Path Halving compression strategy.

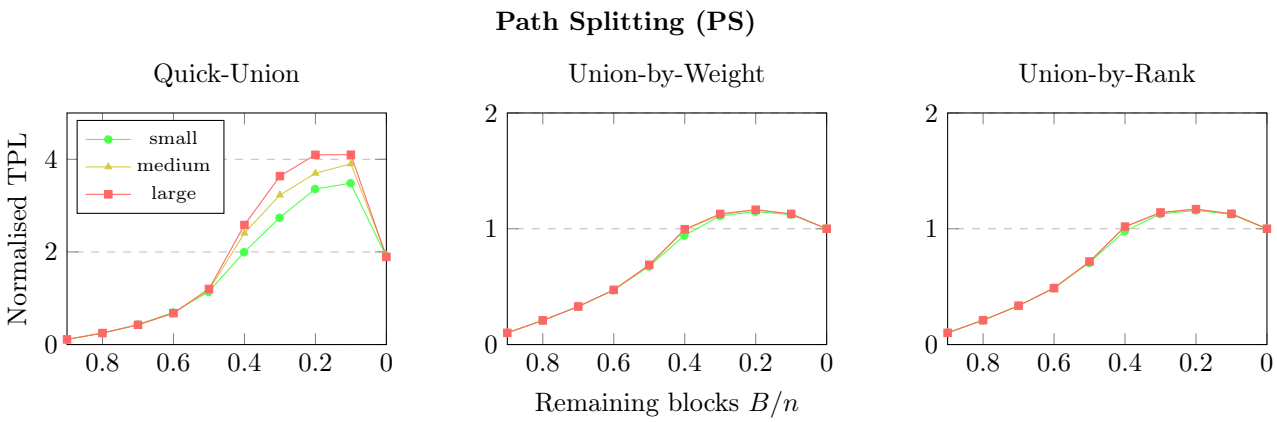

Figure 10: Normalised Total Path Length of the Path Splitting compression strategy.

7.2 Normalised Total Pointer Updates

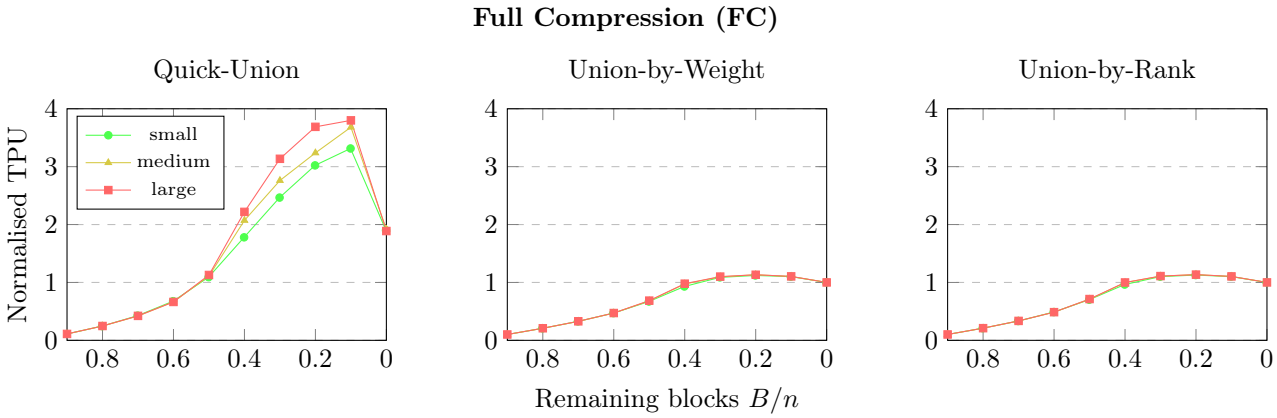

Figure 11: Normalised Total Pointer Updates required with the Full compression strategy.

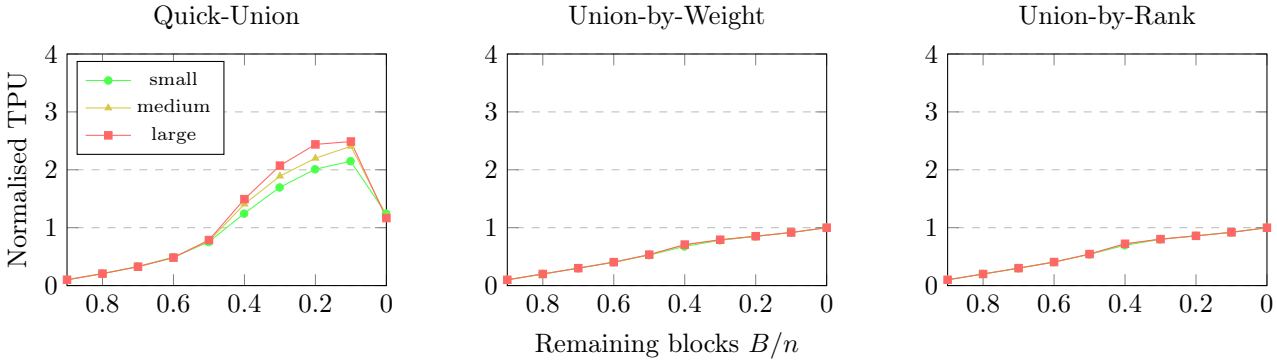

Figure 12: Normalised Total Pointer Updates required with the Path Halving compression strategy.

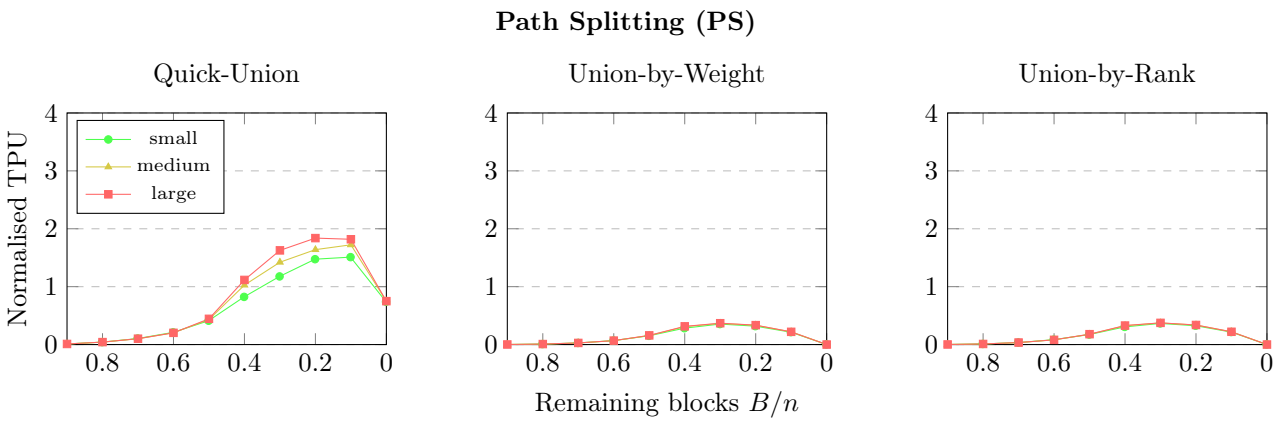

Figure 13: Normalised Total Pointer Updates required with the Path Splitting compression strategy.

7.3 Normalised Cost

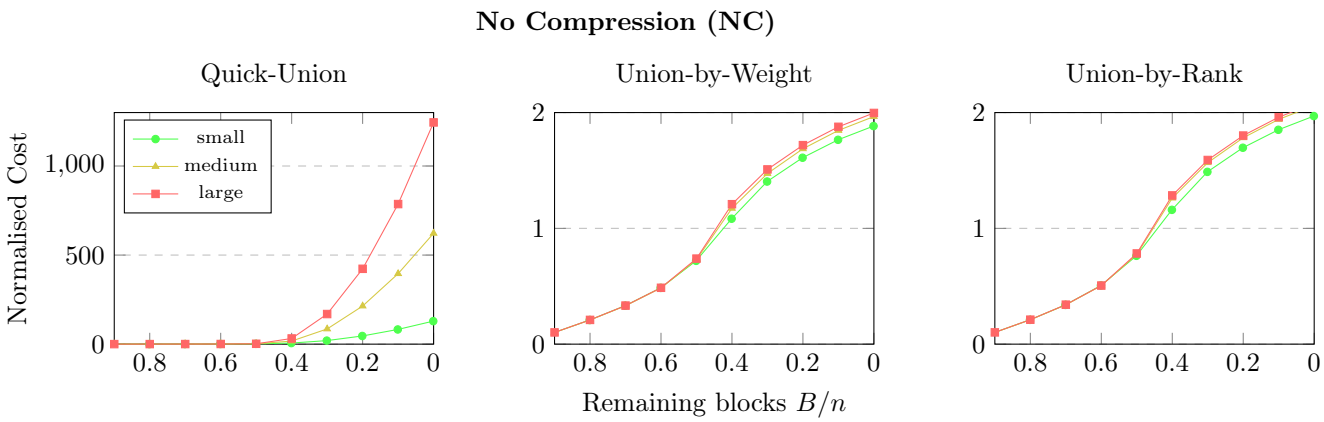

Figure 14: Normalised Cost without compression.

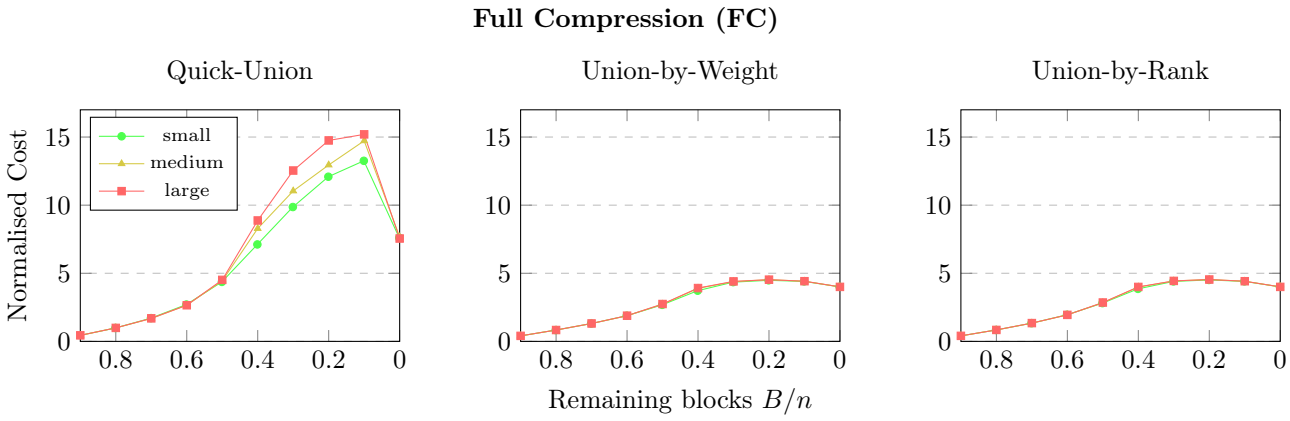

Figure 15: Normalised Cost of the Full Compression strategy.

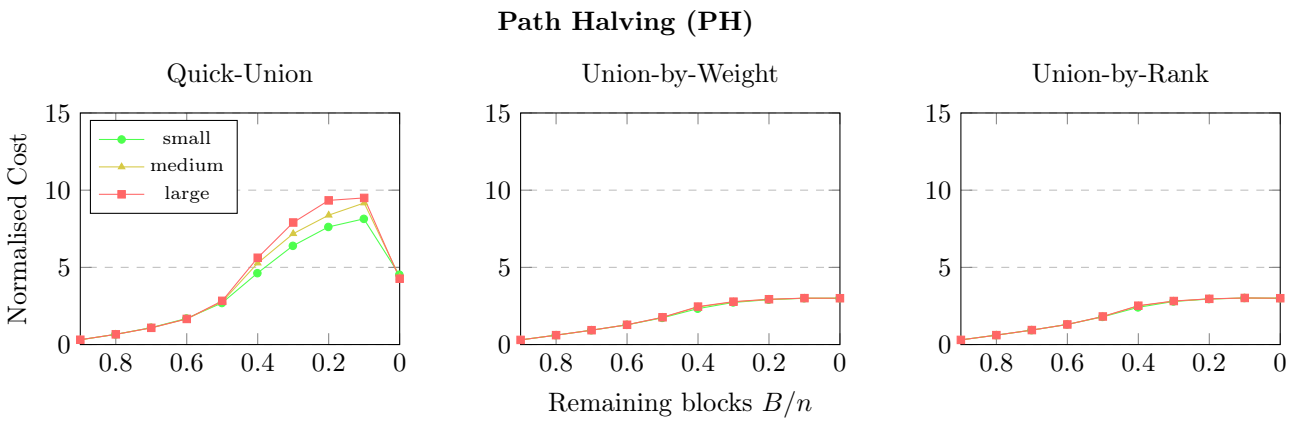

Figure 16: Normalised for Path Halving compression.

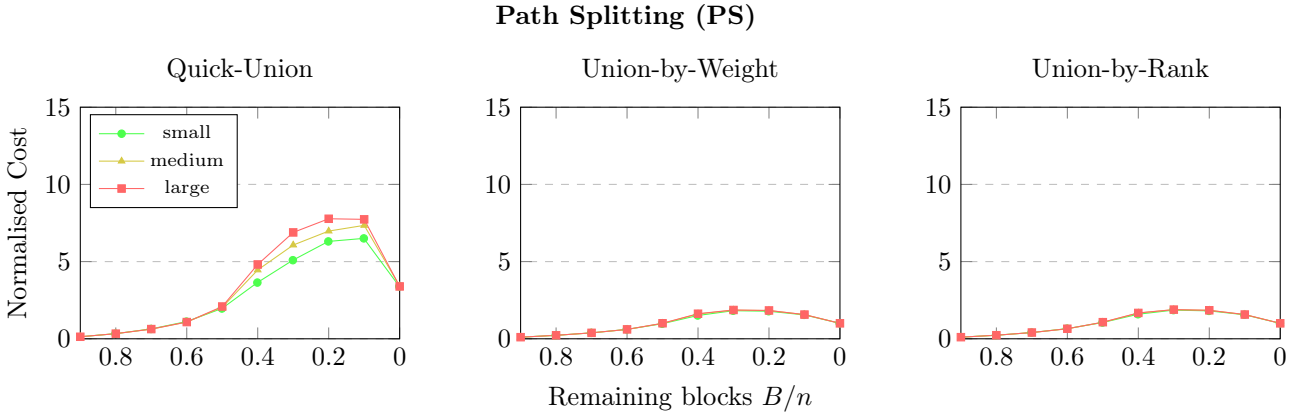

Figure 17: Normalised Cost of the Splitting compression strategy.

Appendix

```cpp
#ifndef UNION_FIND_H
#define UNION_FIND_H

#include "DisjointSet.h"

template<class U, class F>
class UnionFind : public DisjointSet
{
public:
    explicit UnionFind(int n)
        : parent_(n), size_(n,1), rank_(n,0)
    { for (int i=0;i<n;++i) parent_[i]=i; }

    void makeSet(int x) override {
        if (x >= (int)parent_.size()) {
            int old = parent_.size();
            parent_.resize(x+1);
            size_.resize(x+1,1);
            rank_.resize(x+1,0);
            for (int i=old;i<=x;++i) parent_[i]=i;
        } else {
            parent_[x]=x; size_[x]=1; rank_[x]=0;
        }
    }

    int find(int x) override {
        long upd=0;
        return F::find(x,parent_,upd);
    }

    void unionSets(int a,int b) override {
        int r1=find(a), r2=find(b);
        if (r1==r2) return;
        U::unite(r1,r2,parent_,size_,rank_);
    }

    int  getParent(int x) const override { return parent_[x]; }
    int  countSets()      const override {
        int c=0; for (size_t i=0;i<parent_.size();++i) if (parent_[i]==(int)i) ++c; return c;
    }
    int  depth(int x) const override {
        int d=0; while (x!=parent_[x]){x=parent_[x];++d;} return d;
    }
    long pointerUpdatesDuringFind(int x) const override {
        return F::updatesGivenDepth(depth(x));
    }

    static double followMultiplier() { return F::followMult(); }

private:
    std::vector<int> parent_, size_, rank_;
};

#endif
```

Listing 15: `UnionFind.h`

```cpp
#ifndef DISJOINT_SET_H
#define DISJOINT_SET_H

class DisjointSet {
public:
    virtual ~DisjointSet() = default;

    virtual void makeSet(int x)                     = 0;
    virtual int find(int x)                         = 0;
    virtual void unionSets(int x,int y)             = 0;

    virtual int getParent(int x)             const = 0;
    virtual int countSets()                  const = 0;
    virtual int depth(int x)                 const = 0;
    virtual long pointerUpdatesDuringFind(int x) const = 0;
};

#endif
```

Listing 16: `DisjointSet.h`

```cpp
#ifndef UNION_POLICIES_H
#define UNION_POLICIES_H

#include <vector>

struct QU {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    { parent[r1] = r2; }
};

struct UW {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    {
        if (size[r1] < size[r2]) std::swap(r1,r2);
        parent[r2] = r1;
        size[r1] += size[r2];
    }
};

struct UR {
    static void unite(int r1,int r2,
                      std::vector<int>& parent,
                      std::vector<int>& size,
                      std::vector<int>& rank)
    {
        if (rank[r1] < rank[r2]) std::swap(r1,r2);
        parent[r2] = r1;
        if (rank[r1] == rank[r2]) ++rank[r1];
    }
};
#endif
```

Listing 17: `UnionPolicies.h`

```cpp
#ifndef FIND_POLICIES_H
#define FIND_POLICIES_H

#include <vector>

struct NC {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (x != p[x]) x = p[x];
        return x;
    }
    static long updatesGivenDepth(int depth) { return 0; }
    static double followMult() { return 1.0; }
};

struct FC {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        if (p[x] != x) {
            p[x] = find(p[x], p, upd);
            ++upd;
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return depth; }
    static double followMult() { return 2.0; }
};

struct PS {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (p[x] != p[p[x]]) {
            int parent = p[x];
            p[x] = p[parent];
            ++upd;
            x = parent;
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return depth/2; }
    static double followMult() { return 1.0; }
};

struct PH {
    static int find(int x, std::vector<int>& p, long& upd)
    {
        while (p[x] != p[p[x]]) {
            p[x] = p[p[x]];
            ++upd;
            x = p[x];
        }
        return p[x];
    }
    static long updatesGivenDepth(int depth) { return (depth+1)/2; }
    static double followMult() { return 1.0; }
};
#endif
```

Listing 18: `FindPolicies.h`

```cpp
#include "DisjointSet.h"
#include <vector>
#include <random>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <algorithm>
#include <fstream>
#include "util/json.hpp"
#include "UnionPolicies.h"
#include "FindPolicies.h"
#include "UnionFind.h"

using json = nlohmann::json;
struct Metrics { long tpl = 0; long tpu = 0; };

Metrics measureMetrics(const DisjointSet& uf, int n)
{
    Metrics m;
    for (int v = 0; v < n; ++v) {
        m.tpl += uf.depth(v);
        m.tpu += uf.pointerUpdatesDuringFind(v);
    }
    return m;
}

inline std::pair<int,int> indexToPair(long long k)
{
    long long i = static_cast<long long>( (1 + std::sqrt(1 + 8.0*k)) / 2 );
    while (i*(i-1)/2 > k) --i;
    long long j = k - (i*(i-1))/2;
    return {static_cast<int>(i), static_cast<int>(j)};
}

class PairPermutation {
    public:
        PairPermutation(long long m, unsigned seed)
            : perm_(m), idx_(0)
        {
            for (long long i = 0; i < m; ++i) perm_[i] = i;
            std::shuffle(perm_.begin(), perm_.end(), std::mt19937_64(seed));
        }

        bool next(long long &k)
        {
            if (idx_ >= perm_.size()) return false;
            k = perm_[idx_++];
            return true;
        }
    private:
        std::vector<long long> perm_;
        size_t                 idx_;
    };

    using Factory = std::function<std::unique_ptr<DisjointSet>(int)>;

    struct Variant {
        std::string name;
        double      followMult;
        Factory     make;
    };
```

```
    const std::vector<Variant> variants = {
        {"QU_NC", 1.0, [] (int n){ return std::make_unique<UnionFind<QU,NC>>(n); }},
        {"QU_FC", 2.0, [] (int n){ return std::make_unique<UnionFind<QU,FC>>(n); }},
        {"QU_PS", 1.0, [] (int n){ return std::make_unique<UnionFind<QU,PS>>(n); }},
        {"QU_PH", 1.0, [] (int n){ return std::make_unique<UnionFind<QU,PH>>(n); }},
        {"UW_NC", 1.0, [] (int n){ return std::make_unique<UnionFind<UW,NC>>(n); }},
        {"UW_FC", 2.0, [] (int n){ return std::make_unique<UnionFind<UW,FC>>(n); }},
        {"UW_PS", 1.0, [] (int n){ return std::make_unique<UnionFind<UW,PS>>(n); }},
        {"UW_PH", 1.0, [] (int n){ return std::make_unique<UnionFind<UW,PH>>(n); }},
        {"UR_NC", 1.0, [] (int n){ return std::make_unique<UnionFind<UR,NC>>(n); }},
        {"UR_FC", 2.0, [] (int n){ return std::make_unique<UnionFind<UR,FC>>(n); }},
        {"UR_PS", 1.0, [] (int n){ return std::make_unique<UnionFind<UR,PS>>(n); }},
        {"UR_PH", 1.0, [] (int n){ return std::make_unique<UnionFind<UR,PH>>(n); }}
    };

void runExperiment(int n, int delta, int T,
                   bool csv, double followMult, double epsilon,
                   const Factory& makeUF)
{
    const int steps = (n - 1) / delta + 1;
    std::vector<long long> accTPL(steps,0), accTPU(steps,0);
    std::vector<int>     accCnt(steps,0);

    const unsigned baseSeed = 42;
    for (int t = 0; t < T; ++t)
    {

    auto uf = makeUF(n);

    PairPermutation perm(1LL*n*(n-1)/2, baseSeed + t);
    long long k;
    int nextThresh = n - delta + 1;
    int slot      = 0;

    while (perm.next(k)) {
        auto [i,j] = indexToPair(k);
        uf->unionSets(i,j);

        if (uf->countSets() <= nextThresh) {
            Metrics m = measureMetrics(*uf, n);
        accTPL[slot] += m.tpl;
        accTPU[slot] += m.tpu;
        accCnt[slot] += 1;

        nextThresh  = std::max(1, nextThresh - delta);
        ++slot;
        if (uf->countSets() == 1) break;
        }
    }
    }

    if (!csv) {
    std::cout << std::left
        << std::setw(18) << "Number of Blocks"
        << std::setw(15) << "AvgTPL"
        << std::setw(15) << "AvgTPU"
        << std::setw(15) << "Cost"
        << std::setw(15) << "TPL/n"
        << std::setw(15) << "TPU/n"
        << std::setw(15) << "Cost/n" << '\n'
        << std::string(108,'-') << '\n';
```

```
    }

    for (int s = 0; s < steps; ++s) {
    if (accCnt[s] < 5) continue;

    int   blocks = (s < steps-1) ? n - delta + 1 - s*delta : 1;
    double tpl   = double(accTPL[s]) / accCnt[s];
    double tpu   = double(accTPU[s]) / accCnt[s];
    double cost  = followMult * tpl + epsilon * tpu;

    if (csv) {
    if (s == 0)
        std::cout << "Blocks,AvgTPL,AvgTPU,Cost,TPL_per_n,TPU_per_n,Cost_per_n\n";
    std::cout << blocks << ','
           << tpl    << ','
           << tpu    << ','
           << cost   << ','
           << tpl/n  << ','
           << tpu/n  << ','
           << cost/n << '\n';
    } else {
    std::cout << std::left
           << std::setw(18) << blocks
           << std::setw(15) << tpl
           << std::setw(15) << tpu
           << std::setw(15) << cost
           << std::setw(15) << tpl/n
           << std::setw(15) << tpu/n
           << std::setw(15) << cost/n << '\n';
    }
    }
}

int main(int argc, char* argv[])
{

    int    n         = 1000;
    int    delta     = 100;
    int    T         = 20;
    bool   csv       = false;
    double epsilon   = 2.0;

    if (argc == 2) {
        std::ifstream in(argv[1]);
        if (!in) { std::cerr << "Cannot open config file " << argv[1] << '\n'; return 1; }
        json cfg; in >> cfg;
        if (cfg.contains("n"))       n         = cfg["n"];
        if (cfg.contains("delta"))   delta     = cfg["delta"];
        if (cfg.contains("T"))       T         = cfg["T"];
        if (cfg.contains("csv"))     csv       = cfg["csv"];
        if (cfg.contains("epsilon")) epsilon   = cfg["epsilon"];
    }

for (const auto& v : variants) {
    std::cout << "\n=== " << v.name << " ===\n";
    runExperiment(n, delta, T, csv, v.followMult, epsilon, v.make);
}
    return 0;
}
```

Listing 19: `main.cpp`

```
CXX      = g++
CXXFLAGS = -std=c++17 -O2 -Wall -Iinclude
TARGET   = benchmark

SRCS = main.cpp
OBJS = $(SRCS:.cpp=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CXX) $(CXXFLAGS) -o $@ $^

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

clean:
    rm -f $(TARGET) $(OBJS)

run: $(TARGET)
    ./$(TARGET)
```

Listing 20: `Makefile`