University of Copenhagen

Data-Parallel Union-Find

Author: William Henrich Due

1 Theory

This section will derive the theoretical foundation for the data-parallel union-find. The theory is based on set theory, graph theory and properties of forests.

1.1 Forests

To be able to define a union-find structure we first need to define some basic graph theory and specifically in relation to forests. The graphs used are directed graphs $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. Initially we need a definition of what it means for node $u \in V$ to be reachable from node $v \in V$ meaning there is a path from u to v.

Definition 1.1 (Reachability)**.** A node v is *reachable* from a node u in a directed graph $G = (V, E)$ if there exists a sequence of directed edges $e_1, e_2, \ldots, e_m \in E$ where $m \geq 1$ and $e_i = (v_{i-1}, v_i)$ for $1 \leq i \leq m$, such that $v_0 = u$ and $v_m = v$. We denote this by $u \rightsquigarrow v$.

One of the first properties of reachability is it is neither reflexive nor *irreflexive*. The reason for chosing such a definition is that the definition of an cycle in a directed graph becomes simply.

Definition 1.2 (Cycle)**.** A cycle in a directed graph $G = (V, E)$ has a cycle if there exists $v \in V$ such that $v \rightsquigarrow v$.

With this definition of cycles a forest can be defined as follows:

Definition 1.3 (Forest)**.** A forest is a directed graph $F = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges such that:

1. There are no cycles $v \not\rightsquigarrow v$ for all $v \in V$, and
2. each node has at most one parent i.e. for all $(u, v_1), (u, v_2) \in E$ it holds that $v_1 = v_2$.

With the definition of a forest we can now define roots in a forest.

Definition 1.4 (Root)**.** A node $v \in V$ in a forest $F = (V, E)$ is a root if it has no parent. This is defined as the predicate:

$$\mathcal{R}_F(v) : v \not\rightsquigarrow u \text{ for all } u \in V$$

Using the definition of a root we can now define a tree as a special case of a forest.

Definition 1.5 (Tree)**.** A tree is a forest $T = (V, E)$ where there exists a unique root $r \in V$ such that $v \rightsquigarrow r$ for all $v \in V \backslash \{r\}$.

Futhermore we will now work towards seeing a forest as a collection of trees. To do this we first need to establish how many roots a forest has.

Proposition 1.1 (Forest Root Count)**.** A forest $F = (V, E)$ where $|V| = n$ and $|E| = n - k$ has k roots.

Proof. Let $F = (V, E)$ be a forest where $|V| = n$ and $|E| = n - k$. By the second property of a forest then $n - k$ vertices must have a parent. Since there are n vertices in total it follows that there are exactly k vertices $r_1, r_2, \ldots, r_k \in V$ that has no parent. Hence there are exactly k roots in F. □

Knowing how many roots a forest does not finish the picture of how a forest is a collection of trees. We also need to show that each vertex also has a path to a root.

Proposition 1.2 (Root Path Exist)**.** In a forest $F = (V, E)$ for each element $v \in V$ there exists a root $r \in V$ such that $\mathcal{R}_F(r)$ and either $v \rightsquigarrow r$ or $v = r$.

Proof. Let $F = (V, E)$ be a forest and $v \in V$ be an arbitrary element in V. By proposition 1.1 there exists at least one root $r \in V$ such that $\mathcal{R}_F(r)$. This can be shown by structural induction on a vertex $v \in V$ that either $v = r$ or $(v, p) \in E$ such that $p \rightsquigarrow r$.

- If $(v, p) \notin E$ then $v \not\rightsquigarrow u$ for all $u \in V$ so $v = r$.
- If $(v, p) \in E$ then by induction hypothesis $p \rightsquigarrow r$ such that $r \in V$ and $\mathcal{R}_F(r)$. It follows that since $v \rightsquigarrow p \rightsquigarrow r$ so $v \rightsquigarrow r$.

□

Lastly we can finish the picture of a forest being a collection of trees by showing that the path from a vertex to a root is unique. So that there is only one tree for each vertex in the forest.

Proposition 1.3 (Unique Path)**.** Let $F = (V, E)$ be a forest and $v, u \in V$. If $v \rightsquigarrow u$ then the path from v to u is unique.

Proof. Let $F = (V, E)$ be a forest, $v, u \in V$ and $v \rightsquigarrow u$. Since every vertex has at most one parent by the second property of a forest it follows that there is only one out going edge from each vertex in the path from v to u. Hence the path from v to u is unique. □

1.2 Union-Find Structure

Using the definition of a forest we can now define a union-find structure, the way it will represented is as a forest. Here the forest represents an equivalence on V where each tree in the forest represents an equivalence class.

Definition 1.6 (Union-Find Structure)**.** A union-find structure is a forest $F = (V, E)$.

The way this equivalence relation is defined is by the representative of each element in the forest. The representative of a node is found by traversing the edges in the forest until a root is found. Using the notion of a root we can now define the representative of an element in a forest.

Definition 1.7 (Representative)**.** The representative of an element $v \in V$ in a forest $F = (V, E)$ is the root $r \in V$ such that there is a path from v to r. This is defined as the function:

$$\rho_F(v) := r \text{ where } r \in V \text{ such that } \mathcal{R}_F(r) \wedge (v \rightsquigarrow r \vee v = r)$$

With the notion of a representative it is possible to define the set of vertices in the same tree in a forest.

Definition 1.8 (Tree Set)**.** The set of vertices of the same tree $\mathcal{E}_F(v)$ in a forest $F = (V, E)$ is defined as:

$$\mathcal{E}_F(v) := \{u : u \in V \text{ where } \rho_F(u) = \rho_F(v)\}$$

The notion of equivalence classes can now be formalized using the the definition of a partition. We need this to show properties of the union-find structure.

Definition 1.9 (Partition)**.** The set $P \subseteq \mathbb{P}(S)$ is a partition of a set S if:

1. $a \neq \emptyset$ for all $a \in P$
2. $a \cap b = \emptyset$ for all $a, b \in P$ where $a \neq b$
3. $\bigcup_{a \in P} a = S$

We say that P is a partition of S and P forms an equivalence relation on S where each $a \in P$ is an equivalence class.

Using the definition of a partition we can now show that a forest is a partition of its vertices based on the tree sets.

Proposition 1.4 (Forest Partition)**.** A forest $F = (V, E)$ is a partition of V for the following set:

$$\{\mathcal{E}_F(v) : v \in V\}$$

Proof. Let $F = (V, E)$ be a forest. We will show that the set in the proposition is a partition of V by showing that it satisfies the three properties in definition 1.9.

1. By definition of $\mathcal{E}_F(v)$ it can not be empty since for $\mathcal{E}_F(v)$ then $\rho_F(v) = \rho_F(v)$. Hence $\mathcal{E}_F(v) \neq \emptyset$ for all $v \in V$.

2. Let a and b be two arbitrary elements in the set such that $a \neq b$. By definition of a and b there exists $v_1, v_2 \in V$ such that $a = \{u : u \in V \wedge \rho_F(u) = \rho_F(v_1)\}$ and $b = \{u : u \in V \wedge \rho_F(u) = \rho_F(v_2)\}$. Since $a \neq b$ it follows that $\rho_F(v_1) \neq \rho_F(v_2)$ since otherwise $a = b$, hence $a \cap b = \emptyset$.

3. Let v be an arbitrary element in V. By proposition 1.2 there exists a root $r \in V$ such that $\mathcal{R}_F(r)$ and $v \rightsquigarrow r$ or $v = r$. By definition of the representative it follows that $\rho_F(v) = r$. Now let $a = \{u : u \in V \wedge \rho_F(u) = \rho_F(v)\}$. By definition of a it follows that $v \in a$. Since v was arbitrary it follows that $\bigcup_{a \in P} a = V$.

□

With the notion of a partition of a forest we can now define the equivalence relation on the forest.

Definition 1.10 (Same Tree Relation)**.** The relation $\sim_F$ on a forest F is defined as:

$$u \sim_F v :\iff u \in \mathcal{E}_F(v)$$

Trivially we can now show that the same tree relation is an equivalence relation.

Corollary 1.1 (Same Tree Relation is an Equivalence Relation)**.** The relation $\sim_F$ on a forest F is an equivalence relation due to $\{\mathcal{E}_F(v) : v \in V\}$ being a partition of V.

Proof. From proposition 1.4 we directly get that $(V, \sim_F)$ is an equivalence relation since the set $\{\mathcal{E}_F(v) : v \in V\}$ is a partition of V. □

Now using the notion of partitions we can determine if two forests are equivalent in the sense that they have the same tree sets. This is useful when showing that two union-find structures are equivalent.

Definition 1.11 (Forests with Equivalent Tree Sets)**.** Two forests $F = (V, E)$ and $F' = (V', E')$ have equivalent tree sets $F \cong F'$ if:

- Vertices are the same $V = V'$.
- The tree sets are equivalent $\mathcal{E}_F(v) = \mathcal{E'}_F(v)$ for all $v \in V$.

It is also possible to define the property of uniting two trees in a forest should satisfy. We would want the resulting forest to have that the two trees are now one tree containing all the elements from both trees. And all other trees in the forest should remain unchanged.

Definition 1.12 (Tree Union Property)**.** The tree union of two elements v and u for a forest $F = (V, E)$ is such that $v \sim_{F'} u$ in a new forest $F' = (V', E')$ and F' satify the following properties:

1. $\mathcal{E}_{F'}(v) = \mathcal{E}_{F'}(u) = \mathcal{E}_F(v) \cup \mathcal{E}_F(u)$ and
2. $\mathcal{E}_{F'}(w) = \mathcal{E}_F(w)$ for all $w \in V \backslash (\mathcal{E}_F(v) \cup \mathcal{E}_F(u))$.

It is now possible to define a tree union operation that satisfies the tree union property. It uses the fact that if we make the representative of one tree the parent of the representative of the other tree then all elements in the two trees will have the same representative in the new forest. But we actually only care about the trees still being trees and not the structure of the trees themselves. Hence it does not matter which representative becomes the parent of the other.

Proposition 1.5 (Root Union)**.** Let forest $F = (V, E)$, $p = \rho_F(u)$ be the representative of u and let $q = \rho_F(v)$ be the representative of v where $q \neq p$. Then define F' as:

$$F' := (V, E \cup \{(q, p)\})$$

Then $u \sim_{F'} v$ in F' and F' will satisfy the properties of a tree union.

Proof. Let $F = (V, E)$, $p = \rho_F(u)$ be the representative of u and let $q = \rho_F(v)$ be the representative of v. By definition q will have parent p in F' and since q is a root it has no parent then F' is a forest. Now for all $w \in \mathcal{E}_F(q)$ it holds that $w \rightsquigarrow q$ or $q = w$ and since $q \rightsquigarrow p$ it follows that $w \rightsquigarrow p$. Hence $w \in \mathcal{E}_{F'}(p)$ for all $w \in \mathcal{E}_F(q)$ and trivially $w \in \mathcal{E}_{F'}(p)$ for all $w \in \mathcal{E}_F(p)$ so it follows that $\mathcal{E}_{F'}(v) = \mathcal{E}_{F'}(u) = \mathcal{E}_F(v) \cup \mathcal{E}_F(u)$. Now let $w \in V \backslash (\mathcal{E}_F(v) \cup \mathcal{E}_F(u))$ be an arbitrary element. Since $w \not\rightsquigarrow p$, $w \neq p$, $w \not\rightsquigarrow q$, and $w \neq q$ it follows that w has the same representative in F' as in F hence $\mathcal{E}_{F'}(w) = \mathcal{E}_F(w)$. □

This operation can now be used to define the union operation on a union-find structure. This operation corresponds to the union operation in a trivial sequential union-find data structure. The efficient implementations of union-find will do things like path compression or path halving to optimize the distance from a node to its representative. And these operations would also fulfill the tree union property since they do not change the equivalence classes of the forest.

1.3 Conflict-Free Sets

The problem now is how to define a parallel union operation on a union-find structure. The challenge is that multiple union operations might try to change the same part of the forest at the same time. And do to the nature of working in a data-parallel model we can not have atomic operations that would solve these problems in an concurrent manner. The way this problem will be solve is we consider a set of pairs of vertices that must be unified in the forest. We can find the representative of all of these vertex pairs, these representative pairs forms an graph of roots in the forest. We can simply pick out a subset of edges from this graph of roots that forms a forest and glue these edges onto the original forest. This will ensure that there are no conflicts when adding the edges to the original forest. We will call such a set of edges a conflict-free set. An example of a conflict-set is Root Union 1.5 since it only adds one edge between two roots to the forest and hence there can not be any conflicts.

An example of this process can be seen in figures 1 here we have a forest with three trees. In figure 2 we see the graph of roots on the left and on the right a conflict-free set highlighted in red. Finally in figure 3 we see the resulting forest after adding the conflict-free set to the original forest.

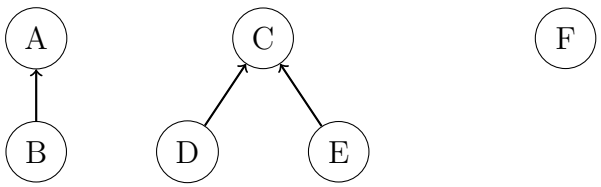

Figure 1: A forest where there are three trees with representatives A, C and F.

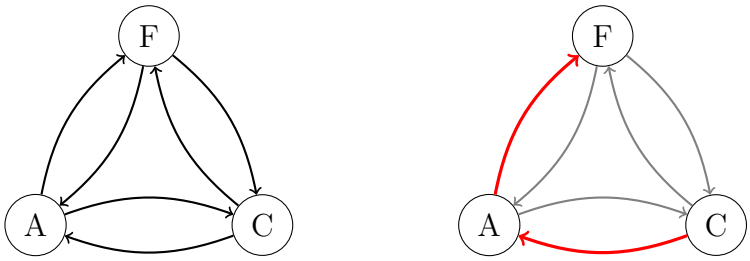

Figure 2: On the left a graph of roots from Figure 1. On the right a conflict-free set highlighted in red.

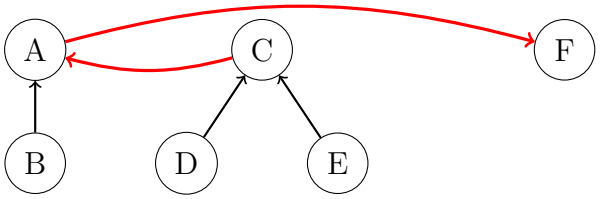

Figure 3: The forest from Figure 1 after adding the conflict-free set from Figure 2.

The formal definition of a conflict-free set is as previous described a set of root pairs that forms a forest.

Definition 1.13 (Conflict-free Set)**.** Let F be a forest, $X \subseteq \{(\rho_F(v), \rho_F(u)) : (v, u) \in V \times V\}$ be a set of root pairs. Then X is a conflict-free set in F if (V, Y) is a forest.

With this definition of a conflict-free set we now wish to show that we can add all edges in a conflict-free set to a forest and still have a forest.

Proposition 1.6 (Conflict-free Forest Union)**.** Let forest $F = (V, E)$ be a forest and let $X \subseteq V \times V$ be a conflict-free set in F where $|X| = n$. Then defining the following forests:

$$F_0 := F$$
$$F_i := (V, E_{i-1} \cup \{(v_i, u_i)\}) \text{ for } (v_i, u_i) \in X \text{ and } 1 \leq i \leq n$$

Then F_n is a forest.

Proof. Let forest $F = (V, E)$ be a forest, $X \subseteq V \times V$ be a conflict-free set in F. We will show that F_n is a forest by induction on i.

- Base case: If $i = 0$ then $F_i = F_0 = F$ which is a forest.

- Induction hypothesis: Assume that F_{i-1} is a forest for all $1 \leq i < n$. Let $(v_i, u_i) \in X$, we know that $v_i \neq v_j$ for all $(v_j, u_j) \in Y \backslash \{(v_i, u_i)\}$ since otherwise (V, X) would not be a forest and X would not be a conflict-free set in F. So v_i will only have one parent in F_i since it only appears once as a child in (V, X). By definition all of the edges in X consists of roots in F, and since (V, X) is a forest there are no cycles $y \not\rightarrow y$ for all $y \in V$ in F_i. Hence F_i is a forest.

Thus by induction F_n is a forest. □

We also need to show that Conflict-free Forest Union 1.6 satisfies Tree Union Property 1.12. This is done by showing that adding edges from the conflict-free set to a forest in any order is equivalent to adding each edge in the same order using Root Union 1.5.

Proposition 1.7 (Conflict-free Set Equivalence)**.** Let forest F be a forest and let $X \subseteq V \times V$ be a conflict-free set in F where $|X| = n$. Then defining the following forests:

$$
\begin{aligned}
F_0 &:= F \\
F_i &:= (V, E_{i-1} \cup \{(v_i, u_i)\}) \text{ for } (v_i, u_i) \in X \text{ and } 1 \leq i \leq n \\
G_0 &:= F \\
G_j &:= (V, E_{i-1} \cup \{(\rho_{G_{j-1}}(v_j), \rho_{G_{j-1}}(u_j))\}) \text{ for } (v_j, u_j) \in X \text{ and } 1 \leq j \leq n
\end{aligned}
$$

Then $F_n \cong G_n$.

Proof. Let forest F be a forest, $X \subseteq V \times V$ be a conflict-free set in F. We will show that $F_n \cong G_n$. We know that for some $(v_i, u_i) \in X$ then $v_i \neq y$ for all $(y, w) \in X \backslash \{(v_i, u_i)\}$ since otherwise (V, X) would not be a forest and X would not be a conflict-free set in F. So all edge set unions will only give a root v_i a new parent u_i once. So $\rho_{F_n}(v_i) = \rho_{F_n}(u_i)$ and $\rho_{G_n}(u_i) = \rho_{G_n}(v_i)$ hence v_i remains in the same tree in both F_n and G_n. Since this holds for all $(v_i, u_i) \in X$ it follows that all elements in V remains in the same tree in both F_n and G_n. Hence $F_n \cong G_n$. □

From this equivalence it will be shown that Conflict-free Forest Union 1.6 satisfies Tree Union Property 1.12.

Corollary 1.2 (Conflict-free Union Satisfies Tree Union Property)**.** Let forest F be a forest and let $X \subseteq V \times V$ be a conflict-free set in F where $|X| = n$. Then defining the following forests:

$$
\begin{aligned}
F_0 &:= F \\
F_i &:= (V, E_{i-1} \cup \{(v_i, u_i)\}) \text{ for } (v_i, u_i) \in X \text{ and } 1 \leq i \leq n
\end{aligned}
$$

Then for all $(v_i, u_i) \in X$ it holds that $v_i \sim_{F_n} u_i$ and F_n satisfies the properties of a tree union.

Proof. Let forest F be a forest, $X \subseteq V \times V$ be a conflict-free set in F. By proposition 1.7 it holds that $F_n \cong G_n$ where G_n is defined as in proposition 1.7. By proposition 1.5 it holds that for all $(v_i, u_i) \in X$ then $v_i \sim_{G_n} u_i$ and G_n satisfies the properties of a tree union. Since $F_n \cong G_n$ it follows that for all $(v_i, u_i) \in X$ then $v_i \sim_{F_n} u_i$ and F_n satisfies the properties of a tree union. □

Now that we have established the properties of conflict-free sets we can now define a method to find a conflict-free set from a set of root pairs. The method chosen to find a conflict-free set is to consider an directed acyclic graph. Such a graph fulfills one property of a forest, namely that there are no cycles. This is ensured by ordering the edges in the graph such that for all edges (v, u) it holds that $v < u$ for some strict total order $(V, <)$. We may do this since the equivalence classes in a forest can be represented by any representative in the tree. Hence we can alwauys pick a total order on the vertices in the forest.

Proposition 1.8 (Ordered Edges Implies Acyclicity)**.** Let $G = (V, E)$ be a directed graph where for all $(v, u) \in E$ it holds that $v < u$ for some strict total order $(V, <)$. Then G has no cycles.

Proof. Let $G = (V, E)$ be a directed graph where for all $(u, v) \in E$ it holds that $u < v$ for some total order $(V, <)$. Let edges $e_1, e_2, \ldots, e_m \in E$ where $m \geq 1$ and $e_i = (v_{i-1}, v_i)$ for $1 \leq i \leq m$ be some path in G. Since the edges are ordered it follows that:

$$v_0 < v_1 < v_2 < \cdots < v_{m-1} < v_m$$

Hence by transitivity of the total order it follows that $v_0 < v_m$. So $v_0 \neq v_m$ hence there are no cycles in G. □

The nice property of using a directed acyclic graph is we can now pick out any vertices from the graph such that no two vertices have the same child. This will ensure that the picked out edges forms a forest.

1.4 Parallel Union-Find

The conflict-free sets makes it is possible to define how unification in the union-find strucutre works. If we consider that we have some forest $F = (V, E)$ and then are given a set of variables pairs $A \subseteq V \times V$ we wish to

unify these pairs such that they are equivalent in some forest F. We can turn A into an acyclic graph (V, Z) where Z only consists of root pairs from F. We want to pick out a subset of Z such that we unify as many of the vertex pairs of Z in F as possible leading to a good time complexity of the algorithm. All of these pairs can not be unified immediately so an algorithm will be first derived which picks out a subset of the directed acyclic graph (V, Z) and then unifies them in F. To determine what a large set is we is first need to define a notion of an edge cover that can be serve as a measure of how many vertex pairs in Z can be unified:

Definition 1.14 (Edge Cover)**.** Let V be a set and $E \subseteq V \times V$ such that $\{v : (v, u) \in E\} \cup \{u : (v, u) \in E\} = V$ then E is an edge cover of V.

Using this definition we know that $V' = \{v : (v, u) \in Z\} \cup \{u : (v, u) \in Z\}$ is an edge cover of the subgraph (V', Z) of (V, Z). Why this is relevant is that vertices in $V \backslash V'$ will not be unified so they are not relevant during unification. Futhermore, the bound that will be establish is for every iteration then atleast $\frac{|V'|}{2}$ vertices must be unified. The intuition behind this is that every time a vertex is given a parent then it can not be given a parent later so it has been dealt with. We can show that when we have such an edge cover then the following inequality holds:

Proposition 1.9 (Edge Cover Inequality)**.** Let V be a set and $E \subseteq V \times V$ be a edge cover of V then:

$$|\{v : (v, u) \in E\}| < \frac{|V|}{2} \implies |\{u : (v, u) \in E\}| > \frac{|V|}{2}$$

Proof. Let V be a set and $E \subseteq V \times V$ be a edge cover of V, and $|\{v : (v, u) \in E\}| < \frac{|V|}{2}$. Let $A = \{v : (v, u) \in E\}$, $B = \{u : (v, u) \in E\}$ and $C = B \backslash A$. By definition of C we have $A \cap C = \emptyset$ so $|A| + |C| = |V|$ and since $|B| \geq |C|$ we can conclude that:

$$|B| \geq |C| = |V| - |A| > |V| - \frac{|V|}{2} = \frac{|V|}{2}$$

Hence $|\{u : (v, u) \in E\}| > \frac{|V|}{2}$. □

This inequality tells us that if Z does not resolve enough vertices, then if we invert the edges in Z then it would be possible to resolve enough vertices. It just remains to show that inverting these edges direction will still give an acyclic graph.

Proposition 1.10 (Inverted Acyclic Graph is Acyclic)**.** Let $G = (V, E)$ be a directed acyclic graph. Then the inverted graph $G' = (V, E')$ where $E' = \{(u, v) : (v, u) \in E\}$ is also acyclic.

Proof. Let $G = (V, E)$ be a directed acyclic graph and $G' = (V, E')$ where $E' = \{(u, v) : (v, u) \in E\}$ is the inverted graph. Let edges $e_1, e_2, \ldots, e_m \in E'$ where $m \geq 1$ and $e_i = (v_{i-1}, v_i)$ for $1 \leq i \leq m$ be some path in G'. By definition of E' it follows that there exists edges $e'_1, e'_2, \ldots, e'_m \in E$ where $e'_i = (v_i, v_{i-1})$ for $1 \leq i \leq m$. If there was a cycle in G' then it would hold that $v_0 = v_m$. But since G is acyclic it follows that $v_0 \neq v_m$. Hence there are no cycles in G'. □

The algorithm which tries to unifies atleast $\frac{|V'|}{2}$ can now be defined as following:

Algorithm 1.1 (Maximal Union)**.** Let forest $F = (V, E)$ be a forest and let $Z \subseteq \{(\rho_F(v), \rho_F(u)) : (v, u) \in V \times V\}$ be a set of root pairs F and (V, Z) is an acyclic directed graph. The maximal union algorithm is defined as:

MaximalUnion(F, Z)

1. $(V, E) \leftarrow F$
2. $V' \leftarrow \{v : (v, u) \in Z\} \cup \{u : (v, u) \in Z\}$
3. $Z \leftarrow \begin{cases} \{(u, v) : (v, u) \in Z\} & \{v : (v, u) \in Z\} < \frac{|V'|}{2} \\ Z & \{v : (v, u) \in Z\} \geq \frac{|V'|}{2} \end{cases}$
4. Let $X \subseteq Z$ where $\{v : (v, u) \in X\} = \{v : (v, u) \in Z\}$ and $|X| = |\{v : (v, u) \in Z\}|$
5. $E \leftarrow E \cup X$
6. `return` $((V, E), Z \backslash X)$

Proposition 1.11 (Maximal Union Correctness)**.** Let forest $F = (V, E)$ be a forest and let $Z \subseteq \{(\rho_F(v), \rho_F(u)) : (v, u) \in V \times V\}$ be a set of root pairs F and (V, Z) is an acyclic directed graph. Then the maximal union algorithm returns a forest $F' = (V, E')$ and a conflict-free set $X \subseteq Z$ in F.

Proof. Let forest $F = (V, E)$ be a forest and let $Z \subseteq \{(\rho_F(v), \rho_F(u)) : (v, u) \in V \times V\}$ be a set of root pairs F and (V, Z) is an acyclic directed graph. Since X is defined such that $\{v : (v, u) \in X\} = \{v : (v, u) \in Z\}$ and $|X| = |\{v : (v, u) \in Z\}|$ it follows that X is a conflict-free set in F since no vertex v appears more than once as a child in X i.e. (V, X) is a forest. By adding the edges in X to E it follows by proposition 1.6 that $F' = (V, E')$ is a forest where $E' = E \cup X$. □

Proposition 1.12 (Left Maximal Union Time Complexity)**.** Let forest $F = (V, E)$ be a forest and let $Z \subseteq \{(\rho_F(v), \rho_F(u)) : (v, u) \in V \times V\}$ be a set of root pairs F and (V, Z) is an acyclic directed graph. Then the maximal union algorithm runs in $O(|Z|)$ work and $O(\log |Z|)$ depth.

Proof. For step 1. it takes $O(1)$ work and $O(1)$ if we assume that E is only used once in this function. Step 2. can be implemented by a parallel sort on the first element of each pair in Z followed by a parallel filter that selects the first occurrence of each unique first element. This takes $O(|Z|)$ work using radix sort with a fixed key length and $O(\log |Z|)$ depth. Step 3. takes $O(|X|)$ work and $O(1)$ depth to add the edges in X to E. Step 4. takes $O(|Z|)$ work and $O(\log |Z|)$ depth to compute the set difference $Z \backslash X$ by a filter. Hence the total work is $O(|Z|)$ and the total depth is $O(\log |Z|)$. Hence the left maximal conflict-free set algorithm runs in $O(|Z|)$ work and $O(\log |Z|)$ depth. □

Algorithm 1.2 (Parallel Tree Union)**.** Let forest $F = (V, E)$ be a tree and let $A \subseteq V \times V$ be a set of pairs of elements in V that will be unioned in parallel. The parallel tree union algorithm is defined as:

ParallelTreeUnion(F, A)

1. $Z_p \leftarrow \{(\rho_F(v), \rho_F(u)) : (v, u) \in A \wedge \rho_F(v) \neq \rho_F(u)\}$
2. $Z \leftarrow \{(\min\{v, u\}, \max\{v, u\}) : (v, u) \in Z_p\}$
3. `while` $|Z| > 0$ `do`
4. $\quad (F, Z_q) \leftarrow$ *LeftMaximalUnion*(F, Z)
5. $\quad Z_r \leftarrow \{(\rho_F(v), \rho_F(u)) : (v, u) \in Z_q \wedge \rho_F(v) \neq \rho_F(u)\}$
6. $\quad Z \leftarrow \{(\min\{v, u\}, \max\{v, u\}) : (v, u) \in Z_r\}$
7. `return` F

Proposition 1.13 (Parallel Tree Union Correctness)**.** Let forest $F = (V, E)$ be a tree and let $A \subseteq V \times V$ be a set of pairs of elements in V that will be unioned in parallel. Then the parallel tree union algorithm returns a forest $F' = (V, E')$ where for all $(v, u) \in A$ it holds that $v \sim_{F'} u$.

Proposition 1.14 (Parallel Tree Union Time Complexity)**.** Let forest $F = (V, E)$ be a tree and let $A \subseteq V \times V$ be a set of pairs of elements in V that will be unioned in parallel. Then the parallel tree union algorithm runs in $O(|A|^2)$ work and $O(|V|)$ span.

1.5 Union by Size and Rank

1.6 Parallel Union-Find Imrovements

2 Implementation

2.1 Interface

The interface of the union-find structure consists of a data-type which is the union-find structure itself. The elements in the union-find structure are represented as integers. These integers are called *handles* and are used to refer to the elements in the union-find structure. The union-find structure is initialized with a fixed number of elements n where the handles are in the range $[0, n-1]$ so they can be used as indices in an array of size n. When exposing these elements to a user then they are abstract datatypes such that the the user cannot do unintened operations on the handles or give invalid handles to the union-find structures operations.

The operations supported are *find* and *union*. The find operation takes a handle and returns the representative which is also a handle. This can be used to check if two elements are in the same set or is "equivalent" by checking if their representatives are the same. The union operation takes two handles such that they have the same representative and are therefore in the same set or to be considered "equivalent". These operations are done in bulk meaning that the find operation takes an array of handles and union can union multiple pairs of handles in parallel.

2.2 Union-find

The first and simplest implementation of the union-find structure is based on the basic parallel tree union algorithm 1.2. It will be extremely inefficient if we do not apply normalization of a left maximal union since you may produce long chains of elements pointing to each other which will make the find operation very expensive. This can be solves by compressing the paths during the find operation such that all elements point directly to their representative after a left maximal union. This will not optimize a sequence of union or find operations but assuming you only have to do one singular bulk union followed by multiple bulk find operations it will be efficient.

2.2.1 Data type

A simple union-find structure can be implemented using an array indices which where the index represents the element/handle and the value at that index is the parent of that element. If the value at that index is a special value which is the highest possible integer value then that element is a root and therefore its own representative. In pseudo code we denote this as an array of integers:

$parents : [n]\mathbf{int}$

Initially all of these values are set to *none* which indicates that all elements are their own representative. Here *none* is defined as an integer $n \geq none$.

2.2.2 Find

To implement the find operation we can simple do a parallel map over all elements to find their representative by a simple loop that follows the parent pointers until a root is found. Since this operation does not modify the structure we can simplify implement it as so:

```
def find_one (parents : [n]int) (handle : int) =
    if parents[handle] = none
    then handle
    else find parents (parents[handle])
```

Then the bulk find operation can be implemented as:

```
def find (parents : [n]int) (handles : [m]int) =
    map (find_one parents) handles
```

2.2.3 Union

The union operation can be implemented using the parallel tree union algorithm 1.2 but due to its abstract nature one must figure out how to make it work in the concrete implementation.

The initial step is to find the representatives of all handle pairs that will be unioned. This can be done by a map followed by a filter to remove pairs where both elements already have the same representative:

```
def find_pairs (parents : [n]int) (pairs : [m](int, int)) =
    let find_one_pair (v, u) =
      (find_one parents v, find_one parents u)
    in (filter (λ(v, u) → v ≠ u) ∘ map find_one_pair) pairs
```

The next step is we want to order all pairs such that it forms an acyclic directed graph. This can be done by simply ordering each pair such that the first element is always less than the second element:

```
def order (pairs : [m](int, int)) =
    map (λ(v, u) → if v < u then (v, u) else (u, v)) pairs
```

Now left maximal union can be performed on the ordered pairs, we do this simply by selecting one parent for each unique first element in the pairs. This can be done by a reduce by index operation using *min* as the reduction operator. Now we just have to partition the pairs into those that were successfully unioned and those that were not. The ones that were not unioned are those where the parent did not change.

There is just one problem with this approach and it is that this may construct one long chain of elements pointing to each other which will make future find operations expensive. This can be solved by normalizing the structure after the left maximal union such that all elements that were given a new parent now point directly to their representative. This results in the following left maximal union implementation:

```
def left_maximal_union (parents : [n]int) (pairs : [m](int, int)) =
    let (l, r) = unzip pairs
    let parents′ = reduce_by_index parents min none l r
    let (remaining, done) = partition (λ(i, p) → parents′[i] ≠ p) pairs
    let parents″ = normalize parents′ (map (λ(x, y) → x) done)
    in (parents″, remaining)
```

The normalization step can be implemented by using the wyllie list ranking algorithm [1, p. 59] to compress the paths in the union-find structure. The modification is instead of checking if the parent does not exists we have to check if it is a root or the grandparent is a root then we are done.

```
def normalize_step (is : [m]int) (parents : [n]int) =
    let next (handle : int) =
      if parents[handle] = none
      then handle
      else if parents[parents[handle]] = none
      then parents[handle]
      else parents[parents[handle]]
    in scatter parents is (map next is)
```

Then we just have to repeat this process $\lceil \log_2 m \rceil$ times to ensure that all paths are compressed:

```
def normalize (parents : [n]int) (is : [m]int) =
  let loop (parents' : [n]int) (is' : [m]int) (count : int) =
    if count = 0
    then parents'
    else normalize_step is' parents'
    in loop parents' ps' (count − 1)
  in loop parents is ⌈log2 m⌉
```

The last step in the parallel tree union algorithm is to swap the pairs such that we can perform left maximal union again on the inverted pairs. This takes the acyclic directed graph formed by the remaining pairs and inverts all edges:

```
def swap (pairs : [n](int, int)) =
  map (λ(v, u) → (u, v)) pairs
```

Now we have all the functions needed to implement the union operation:

```
def union (parents : [n]int) (pairs : [m](int, int)) : [n]int =
  let pairs =
```

2.3 Union by Rank

2.4 Union by Size