CS 433 Automated Reasoning 2021

Lecture 11: Implementing QF_EUF

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2021-08-31

Topic 11.1

Union find - an algorithm for equivalences

Union find

Equivalence classes are usually implemented using union-find data structure

- each class is represented using a tree over its member terms
- root of the tree represents the class
- getClass() returns root of the tree, which involves traversing to the root
- mergeClasses() simply adds the root of smaller tree as a child of the root of larger class

Efficient data-structure: for n pushes, run time is $O(n \log n)$

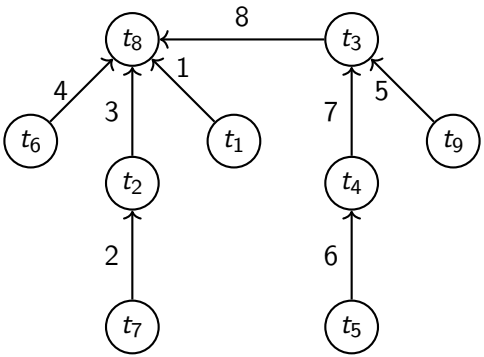
Exercise 11.1

Prove the above complexity

Example: union-find

Example 11.1

Consider: $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



unsatCore using union find

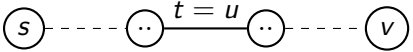
- generate proof of unsatisfiablity using union find
- collect leaves of the proof, which can serve as an unsat core

Proof generation in union-find

Proof generation from union find data structure for an unsat input.
The proof is constructed bottom up.

1. There must be a dis-equality $s \neq v$ that was violated.
   We need to find the proof for $s = v$.

2. Find the latest edge in the path between s and v. Let us say it is due to input literal $t = u$.



   Recursively, find the proof of $s = t$ and $u = v$.

   We stitch the proofs as follows

$$\frac{\dfrac{\dots}{s = t} \quad t = u \quad \dfrac{\dots}{u = v}}{s = v}$$

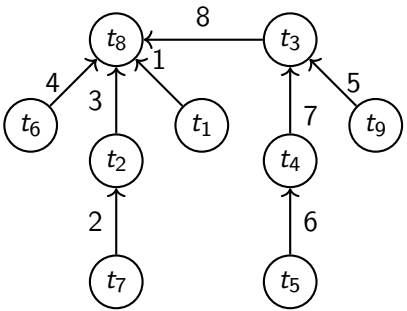
For improved algorithm: R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. RTA'05, LNCS 3467

Commentary: We may need to apply symmetry rule to get the equality in right order.

Example: union-find proof generation

Example 11.2

Consider: $\underbrace{t_1 = t_8}_{1} \wedge \underbrace{t_7 = t_2}_{2} \wedge \underbrace{t_7 = t_1}_{3} \wedge \underbrace{t_6 = t_7}_{4} \wedge \underbrace{t_9 = t_3}_{5} \wedge \underbrace{t_5 = t_4}_{6} \wedge \underbrace{t_4 = t_3}_{7} \wedge \underbrace{t_7 = t_5}_{8} \wedge \underbrace{t_1 \neq t_4}_{9}$



$$\dfrac{\dfrac{\dfrac{}{t_7 = t_1}}{t_1 = t_7} \quad t_7 = t_5 \quad t_5 = t_4}{t_1 \neq t_4 \qquad t_1 = t_4}$$
$$\bot$$

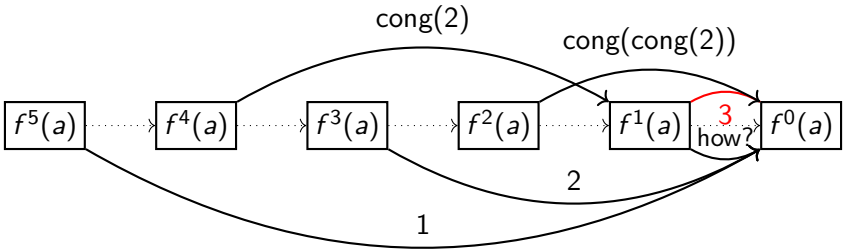
1. $t_1 \neq t_4$ is violated.
2. 8 is the latest edge in the path between t_1 and t_4
3. 8 is due to $t_7 = t_5$
4. Look for proof of $t_1 = t_7$ and $t_5 = t_4$
5. 3 is the latest edge between t_1 and t_7, which is due to $t_7 = t_1$.
6. Similarly, $t_5 = t_4$ is edge 6

Example: extending to congruence

Example 11.3

Run union find on $\underbrace{f^5(a) = a}_{1} \wedge \underbrace{f^3(a) = a}_{2} \wedge \underbrace{f(a) \neq a}_{3}$

$\cdots\cdots\rightarrow$ Term parent relation



Exercise 11.2

Extract proof from the above graph?

Topic 11.2

Union-find in the context of SMT solver

Union-find in the context of SMT solver

SMT solver design causes frequent calls to getClass(), which is not constant time.

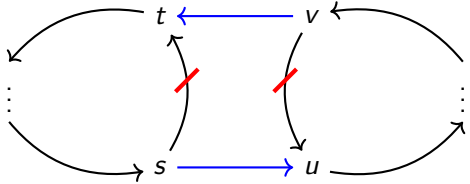
To make it constant time, we may add another field in each node that points to the root.

- Increases the cost of merge: needs to update the root field in each node
- Traversal in the tree needs a stack

Why not use a simpler data structure?

Union-find using circular linked lists

- We may represent the equivalence class using circular linked lists
- each node has a field to indicate the root, therefore getClass() is constant time
- merging two circular linked lists via field *next*



$s.next, v.next := v.next, s.next$

Exercise 11.3

How to split circular linked lists at two given nodes?

Commentary: Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52 (2005) (p386-389,p420-433)

Fields for equivalence classes

Therefore, we need the following fields in nodes to implement equivalence classes over the nodes

- next (pointers for the circular link list)
- root (every node points to the root in the list)
- size (for choosing next root)

Merge/unmerge classes

- On class merge,
  - the two circular linked lists are merged and
  - the root fields in the smaller of the two are set to the root of the other.
  - the "looser" root of the smaller list is recorded in order for possible unmerge

- On backtracking, we iterate over the loosers record in the reverse order and unmerge
  1. Let node x be the current top looser root.
  2. r := getClass(x); $r.next, x.next := x.next, r.next$.
  3. make x root of the part that contains x.

Exercise 11.4

In the above code, we have not written code for updating size fields. Complete the code.

Topic 11.3

Implementing congruence-closure

Congruence-closure

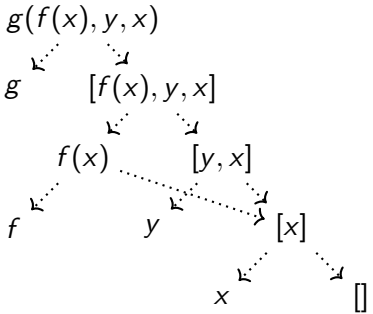
- We need to implement congruence-closure with equality reasoning.
- A uniform data structure to represent function with arbitrary parameters
- Quick way to find application of congruence.

Uniform representation of terms.

Terms as binary DAGs

- Term has two children: Left child is the top symbol and right child is the argument list
- Argument list has two children: left child is the first term and right child is the tail list

Example 11.4



Exercise 11.5

Prove: each class consists of nodes that are either left children or right children.

Equivance classes over terms and lists

We compute equivalence of terms as well as term lists.

We also maintain the following equivalence classes.

1. LeftEquiv: nodes whose left children are in same class
2. RightEquiv: nodes whose right children are in same class
3. BothEquiv: nodes whose left children are in same class and right children are in same class

The above three are equivalence relations. We implement them similarly using circular linked lists by adding fields in the node for each of the relations.

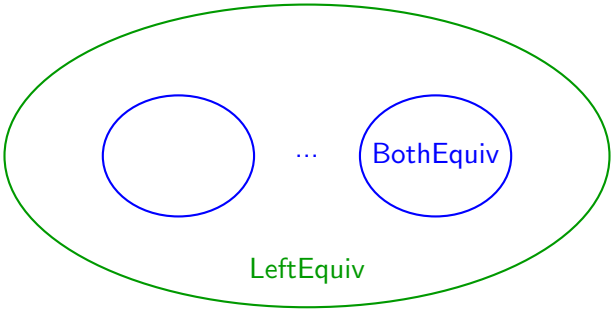
Exercise 11.6

Write all the fields in the node for the above equivalence relations?

Commentary: LeftEquiv and RightEquiv relations are again maintained as circular linked lists. Similarly (un)merged trigger by (un)merger of their children. The looser root needs to keep sufficient information for unmerge. BothEquiv is stored as tree-like union-find data structure.(why?)
Read: Detlefs et.al. Simplify: a theorem prover for program checking, 2005.

Understanding LeftEquiv, RightEquiv, and BothEquiv

Let us consider the case of LeftEquiv

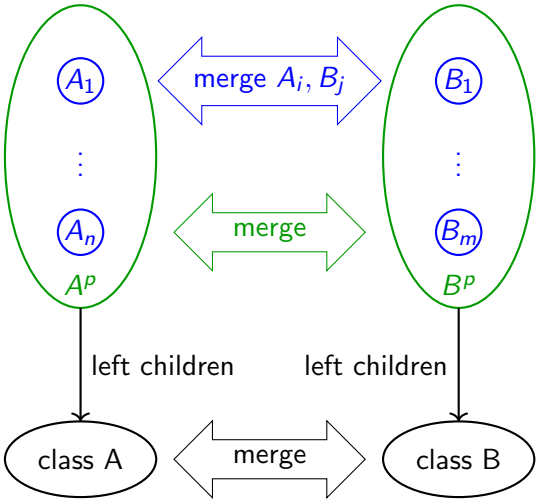


Each LeftEquiv class contains partitions of nodes that form BothEquiv classes.

Similarly, RightEquiv classes contain BothEquiv partitions.

Applying congruence upon merger

1. Classes A and B are being merged.
2. Get parent LeftEquiv classes A^p and B^p
3. Let $A_1, ..., A_n$ be the congruent partitions of A^p
4. Let $B_1, ..., B_m$ be the congruent partitions of B^p
5. Merge A_p and B_p
6. if A_i and B_j are in same RightEquiv class
   - Merge BothEquiv A_i and B_j
   - Congruence found : add $A_i.root = B_j.root$



Enumerating A_i and B_j at step 6 is expensive. We may save some time using a hash map.

Commentary: At step 2, A and B were left children of their respective parents. We may also have a case where A and B were right children.

Congruence table

We maintain a hash map $\mathit{CongTable} : \mathit{LeftEquiv} \times \mathit{RightEquiv} \hookrightarrow \mathit{BothEquiv}$, which records if a node belongs to $X \in \mathit{LeftEquiv}$ and $Y \in \mathit{RightEquiv}$, then it must belong to $\mathit{CongTable}(X, Y)$.

Using CongTable,

- Wlog, let A_p be smaller than B_p.
- For each A_i, let A_i^r be $\mathit{RightEquiv}$ of A_i.
- If $B_j = \mathit{CongTable}(A_i^r, B_p)$ exists, then we merge A_i and B_j.

Using Hash table, we need to enumerate only one set.

Maintaining CongTable: we need to update $\mathit{CongTable}$ on each merge and unmerge operations.

Commentary: For the details for maintaining *CongTable* read: Detlefs et.al. Simplify: a theorem prover for program checking, 2005. Section 7

Example: congruence data structure

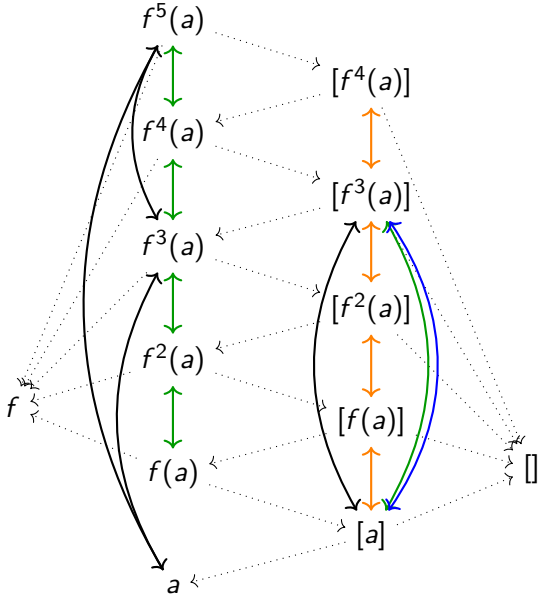
Example 11.5

Consider $f^5(a) = a \wedge f^3(a) = a$

- *Term graph is denoted by dotted edges*
- *Initially,*
  $\{f(a), f^2(a), f^3(a), f^4(a), f^5(a)\} \in$ *LeftEquiv (why?)*
  $\{[a], [f(a)], [f^2(a)], [f^3(a)], [f^4(a)]\} \in$ *RightEquiv (why?)*
- *Black edges are equivalence classes*
- $[f^3(a)]$ *and* $[a]$ *form an LeftEquiv class because their left children are equivalent*
- *Since* $[f^3(a)]$ *and* $[a]$ *are in same RightEquiv, they also form an BothEquiv class*
- *Therefore, we add* $[f^3(a)] = [a]$

Exercise 11.7

Complete the run



Topic 11.4

Handling disequality

Data structure for disequalities

For each equivalence class, we maintain a set of the other unmergable classes

- the set cannot be maintained as a circular linked lists over nodes by adding new field
- The set is maintained in a list for which we need extra memory

Exercise 11.8

If we have input that says some $n > 2$ terms are distinct,

$$\texttt{(distinct t1 ... tn)}$$

How many entries we need in the unmergable classes lists?
Can we do it better? *Hint: use bitvectors for each distinct*

Topic 11.5

Problems

Exercise

Exercise 11.9

Apply union-find on the following equalities and draw the resulting tree.

$$\underbrace{t_6 = t_2}_{1} \wedge \underbrace{t_7 = t_4}_{2} \wedge \underbrace{t_8 = t_5}_{3} \wedge \underbrace{t_3 = t_7}_{4} \wedge \underbrace{t_4 = t_3}_{5} \wedge \underbrace{t_5 = t_6}_{6} \wedge \underbrace{t_3 = t_8}_{7}$$

Each equality has been assigned a number. Please label the edges of the tree with the numbers.

Problem

Exercise 11.10

Prove/Disprove that the following formula is unsat.

$$(f^4(a) = a \vee f^6(a) = a) \wedge f^3(a) = a \wedge f(a) \neq a$$

If unsat give a proof otherwise give a satisfying assignment.

Please show a run of DPLL(T) and union-find on the above example.

End of Lecture 11