**Watched Literals**

According to the handbook of satisfiability the watched literals data structure is organized as follows:

1- For each clause, exactly two unassigned literals are watched, using two watch references.
2- The clause's state is only updated when one of the watched literals is assigned.
    a. If the watched literal is satisfied (we say the literal becomes a True literal), nothing more is done with respect to this clause.
    b. If the literal is falsified, then the clause is traversed in search for one non-false literal that is not already being watched.
        i. This results in finding a non-watched unassigned literal that from now on it should watch the clause and currently falsified literal will not watch that clause.
        ii. If no such a literal is found, then it means, the clause is unit. Thus, it should be added to the propagation queue.
3- When backtracking, the watched literals information is not updated.

For implementation, there are some tricks that MiniSAT uses.

First, we define a list of lists named 'watches'. The length of the 'watches' is equal to two times of number of variables. Entry 0 is responsible for keeping the watch list of the literal 1. Entry 0 is responsible for keeping the watch list of the literal -1 (even though the original formula might not have such a literal). Entry 2, and 3 are responsible for keeping the watch list of the literal 2, and -2 respectively, and so on. For convenience, we define a function named 'index' that given a literal it returns the index of the 'watches' we should access to get the watched literals of that literal. Therefore, for example, index(-2) returns 3. Moreover, watches[index(1)] gives the list of clauses that literal 1 is currently watching.

For the sake of convenience, we also contract to make sure the two watched literals of a clause are the first and second literals of it. This does not contradict with what we said that one can choose two arbitrary literals from a clause to be watched. If the two chosen literals are not the two first elements of the clause, swapping their location with the first two elements would resolve the issue. For example, consider a clause like [1, -4, 5, 3, -2] and assume we want to choose 5, and -2 as watch literals. In this case, it is enough to swap them in a way that 5, and -2 would be the first two elements of the clause with whatever order, for example, [-2,5, 3,1, -4]. In practice, we would like to keep changes minimal so, a minimal rearrangement like [5,-2,-4,1,3,-4] would be used in practice.

As a special case, there might be some unit clauses in the original formula. For these clauses there are no two literals to be watched. For these clauses, we do not need to watch anything. In fact, in the initialization step we should identify these clauses and enqueue them into the trail (trail is simultaneously used as a stack for keeping the partial assignments and a propagation queue to queue unit facts; older implementations might use a trail as a stack and a separate queue for propagation). Then, in the unit propagation step, all such facts will be propagated and so all these unit facts will be satisfied. Since watch literals data structure only cares about not yet satisfied clauses, there would be no need to care about these unit clauses consequently.

One important trick that MiniSAT uses is related to where that it stores watches. Consider a clause like c=[5,-2,-4,1,3,-4] that its first two elements are those we want to use for watching the clause. Intuitively, we would write a code like watches[index(c[0])].push(c) and watches[index(c[1])].push(c), and depends on the implementation there is nothing wrong with this. However, MiniSAT uses something differently. It stores watches as watches[index(-c[0])].push(c) and watches[index(-c[1])].push(c). The reason is because of the way it implements the function propagate. As we mentioned, watching literal data structure only cares about not yet satisfied clauses. In the propagation step, when we

dequeue a unit information from the propagation queue, we must make that literal True. Therefore, in that clause all clauses that this literal happens no changes will be needed. However, some changes in clauses where complement of the literals happened might be needed (Note, it is not possible to have a literal and its complement in the same clause). Therefore, to reduce the amount of code needed to access such clauses with potential of having need to change 'watches' and rearranging themselves, MiniSAT uses this way for storing watches.

Finally, MiniSAT uses a clever way of removing watch literals. In the propagation step, it defines two integers i, and j. While i is just a pointer to the current clause under processing, j follows i and it aims to serve for identification of the location were removing should be happened. In fact, when loops are finished, i-j last items of the watch literals of a watches[x] must be removed, as those items do not watch the clauses that have been put at the end of the list.

Overall, with some small differences, Cadical uses the same manner for implementing watched literals. However, at least one important difference is that Cadical implements watched literals optimal while MiniSAT's implementation is not optimal. In fact, Cadical uses a circular watched literal. One can change MiniSAT's watch literal implementation to the circular one. We can add an 'int' property named 'watch' to the 'Clause' datatype of the MiniSAT and initiate it with value 1. Then, in the propagation step, we can use 'watch' to implement circular watched literals. The code made available by the paper that proposed this approach.