Part IV
Minimizing *DFAs*

An interesting *DFA* minimization function is the one due to Brzozowski (appearing in [Wat93b, Section 2]).

User function: *DFA::min_Brzozowski*

Files: `dfa.h`

Uses: see the entry for class *DFA*

Description: Brzozowski's minimization algorithm differs from all of the other minimization algorithms, and appears as inline *DFA* member function *min_Brzozowski*. The member function simply minimizes the *DFA*.

Implementation: The implementation appears in file `dfa.h`. The implementation follows directly from [Wat93b, Section 2].

□

The remainder of Part IV deals with the other minimization functions, based upon their derivations in [Wat93b].

17 Helper functions

Two member functions of class *DFA* are used as helpers in those minimization functions which are based upon an equivalence relation (see [Wat93b, Section 3]).

Implementation function: *DFA::compress*

Files: `dfa.cpp`

Uses: see the entry for class *DFA*

Description: Member function *compress* is overloaded to take either a *StateEqRel* (an equivalence relation on *States*), or a *SymRel* (a symmetrical relation on *States*). The relation is assumed to be a refinement of relation E [Wat93b, Definition 3.3]. The member function implements function *merge* defined in [Wat93b, Transformation 3.1].

Implementation: The implementation appears in file `dfa.cpp`. The implementation follows directly from [Wat93b, Transformation 3.1].

□

Implementation function: *DFA::split*

Files: `min.cpp`

Uses: *CharRange, DFA, State, StateEqRel, StateSet*

Description: Member function *split* takes two *States* (p and q), a *CharRange* a, and a *StateEqRel*. The two *States* are assumed to be representatives in the *StateEqRel*. The equivalence class of p is then split into those *States* (equivalent to p) which transition on a to a *State* equivalent to q, and those that do not. If in fact there was a split (the equivalence class of p is split into two equivalence classes), p will still be the unique representative of one of the two new equivalence classes; in this case, the (unique) representative of the other equivalence class is returned. If no such split occurred, function *split* returns *Invalid*.

Implementation: The implementation is a simple, iterative one. It implements the equivalent of the assignment to Q'_0 in Algorithm 4.4 of [Wat93b].

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:06 $
#include "charrang.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

State DFA::split( const State p, const State q, const CharRange a, StateEqRel& P ) const {
        assert( class_invariant() );
        assert( P.class_invariant() );

        // p and q must be representatives of their eq. classes.
        assert( p == P.eq_class_representative( p ) );
        assert( q == P.eq_class_representative( q ) );

        // Split [p] with respect to [q] and CharRange a.
        auto StateSet part;
        part.set_domain( Q.size() );

        // Iterate over [p], and see whether each member transitions into [q]
        // on CharRange a.
        auto State st;
        for( P.equiv_class( p ).iter_start( st );
```

```
                !P.equiv_class( p ).iter_end( st );
                P.equiv_class( p ).iter_next( st ) ) {
            auto State dest( T.transition_on_range( st, a ) );
            // It could be that dest == Invalid.
            //          if not, check if dest is in [q].
            if( dest != Invalid && P.equivalent( dest, q ) ) {
                part.add( st );
            }
        }

        // The following containment must hold after the splitting.
        assert( P.equiv_class( p ).contains( part ) );

        // Return non-zero if something was split.
        if( (part != P.equiv_class( p )) && !part.empty() ) {
            // Now figure out what the other piece is.
            auto StateSet otherpiece( P.equiv_class( p ) );
            otherpiece.remove( part );
            assert( !otherpiece.empty() );

            P.split( part );
            assert( class_invariant() );

            // Now, we must return the representative of the newly created
            // equivalence class.
            auto State x( P.eq_class_representative( otherpiece.smallest() ) );
            assert( x != Invalid );
            // It could be that p is not in part.
            auto State y( P.eq_class_representative( part.smallest() ) );
            assert( y != Invalid );
            assert( x != y );

            if( p == x ) {
                // If p is the representative of 'otherpiece', then
                // return the representative of 'part'.
                assert( otherpiece.contains( p ) );
                return( y );
            } else {
                // If p is the representative of 'part', then return the
                // representative of 'otherpiece'.
                assert( part.contains( p ) );
                return( x );
            }
        } else {
            assert( (part == P.equiv_class( p )) || (part.empty()) );
            // No splitting to be done.
            return( Invalid );
        }
}
```

□

18 Aho, Sethi, and Ullman's algorithm

User function: *DFA::min_dragon*

Files: `min-asu.cpp`

Uses: *CRSet, DFA, State, StateEqRel, StateSet*

Description: Member function *min_dragon* is an implementation of Algorithm 4.6 in [Wat93b]. It is named the "dragon" function after Aho, Sethi, and Ullman's "dragon book" [ASU86].

Implementation: The algorithm begins with the approximation E_0 to the *State* equivalence relation E. It then repeatedly partitions the equivalence classes until the greatest fixed point (E) is reached.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:58:59 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

// Implement Algorithm 4.6 of the Taxonomy.
DFA& DFA::min_dragon() {
        assert( class_invariant() );
        assert( Usefulf() );

        // We'll need the combination of all of the out-transitions of all of the
        // States, when splitting equivalence classes.
        auto CRSet C;
        auto State st;
        for( st = 0; st < Q.size(); st++ ) {
                C.combine( T.out_labels( st ) );
        }

        // P is the equivalence relation approximation to E. It is initialized
        // to the total relation with domain Q.size().
        auto StateEqRel P( Q.size() );
        // We now initialize it to E_0.
        P.split( F );

        // reps is the set of representatives of P.
        auto StateSet reps( P.representatives() );

        // [st] is going to be split w.r.t. something.

        // The following is slightly convoluted.
        reps.iter_start( st );
        while( !reps.iter_end( st ) ) {
                // Try to split [st] w.r.t. every class [q].
                auto State q;

                // Keep track of whether something was indeed split.
                //        Having to use this variable could be avoided with a goto
                auto int something_split( 0 );

                // Iterate over all q, and try to split [st] w.r.t. all other
                // equivalence classes [q].
                // Stop as early as possible.
                for( reps.iter_start( q ); !reps.iter_end( q )
                                && !something_split; reps.iter_next( q ) ) {
                        // Iterate over the possible transition labels, and
                        // do a split if possible.
                        auto int i;
```

```
                for( i = 0; !C.iter_end( i ) && !something_split; i++ ) {
                        something_split = (split( st, q, C.iterator( i ), P ) != Invalid);
                }
        }

        // If something was split, restart the outer repetition.
        if( something_split ) {
                // The set of representatives will have changed due to
                // the split.
                reps = P.representatives();
                reps.iter_start( st );
                // Now continue the outer repetition with the restarted
                // representatives.
        } else {
                // Just go on as usual.
                reps.iter_next( st );
        }
}

compress( P );
assert( class_invariant() );

return( *this );
}
```

□

19 Hopcroft and Ullman's algorithm

User function: *DFA::min_HopcroftUllman*

Files: `min-hu.cpp`

Uses: *CRSet, DFA, State, StateSet, SymRel*

Description: This member function provides a very convoluted implementation of Algorithm 4.7 of [Wat93b]. It computes the distinguishability relation (relation D in [Wat93b]).

Implementation: The algorithm is not quite the same as that presented in [Wat93b, Algorithm 4.7]. The member function computes the equivalence relation (on *States*) by first computing distinguishability relation D. Initially, the pairs of distinguishable *States* are those pairs p, q where one is a final *State* and the other is not. Pairs of *States* that reach p, q are then also distinguishable (according to the efficiency improvement property given in [Wat93b, Section 4.7, p. 16]). Iteration terminates when all distinguishable *States* have been considered.

Performance: This algorithm can be expected to run quite slowly, since it makes use of reverse transitions in the deterministic transition relation (*DTransRel*). The transition relations (*TransRel* and *DTransRel*) are optimized for forward transitions.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:05 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "symrel.h"
#include "dfa.h"

// Implement (a version of) Algorithm 4.7 of the minimization Taxonomy.
DFA& DFA::min_HopcroftUllman() {
        assert( class_invariant() );
        assert( Usefulf() );

        // We need the combination of all transition labels, to iterate over
        // transitions.
        auto State st;
        auto CRSet C;
        for( st = 0; st < Q.size(); st++ ) {
                C.combine( T.out_labels( st ) );
        }

        // First, figure out which are the non-final States.
        auto StateSet nonfinal( F );
        nonfinal.complement();

        // Use Z to keep track of the pairs that still need to be considered for distinguishedness.
        auto SymRel Z;
        Z.set_domain( Q.size() );

        // We begin with those pairs that are definitely distinguished.
        //        this includes pairs with different parity.
        Z.add_pairs( F, nonfinal );

        // It also includes pairs with differing out-transitions.
        //        iterate over C (the CRSet) and check the haves and have-nots
        auto StateSet have;
        have.set_domain( Q.size() );
        auto StateSet havenot;
        havenot.set_domain( Q.size() );

        auto int it;
```

```
for( it = 0; !C.iter_end( it ); it++ ) {
        // have and havenot must be empty for the update to be correct.
        assert( have.empty() && havenot.empty() );
        auto State p;
        // Iterate over the States and check which have a transition.
        for( p = 0; p < Q.size(); p++ ) {
                // Does p have the transition?
                if( T.transition_on_range( p, C.iterator( it ) ) != Invalid ) {
                        have.add( p );
                } else {
                        havenot.add( p );
                }
        }
        // have and havenot are distinguished from one another.
        //        (under the assumption that Usefulf() holds)
        Z.add_pairs( have, havenot );

        have.clear();
        havenot.clear();
}

// G will be use to accumulate the (distinguishability) relation D.
auto SymRel G;
G.set_domain( Q.size() );

// Now consider all of the pairs until nothing changes.
while( 1 ) {
        auto State p;
        // Go looking for the next pair to do.
        for( p = 0; p < Q.size() && Z.image( p ).empty(); p++ );

        // There may be nothing left to do.
        if( p == Q.size() ) {
                break;
        } else {
                assert( !Z.image( p ).empty() );
                // Choose q such that {p,q} is in Z.
                //        We know that such a q exists.
                auto State q( Z.image( p ).smallest() );
                assert( q != Invalid );

                // Move {p,q} from Z across to G.
                Z.remove_pair( p, q );
                G.add_pair( p, q );

                // Now, check out the reverse transitions from p and q.
                auto int i;
                // Iterate over all of the labels.
                for( i = 0; !C.iter_end( i ); i++ ) {
                        auto StateSet pprime( T.reverse_transition( p,
                                        C.iterator( i ) ) );
                        auto StateSet qprime( T.reverse_transition( q,
                                        C.iterator( i ) ) );
                        // pprime and qprime are all distinguished.
                        // Iterate over both sets and flag them as distinguished.
                        auto State pp;
                        for( pprime.iter_start( pp ); !pprime.iter_end( pp );
                                        pprime.iter_next( pp ) ) {
                                auto State qp;
                                for( qprime.iter_start( qp );
                                                !qprime.iter_end( qp );
                                                qprime.iter_next( qp ) ) {
                                        // Mark pp, qp for processing if they
                                        // haven't already been done.
                                        if( !G.contains_pair( pp, qp ) ) {
                                                Z.add_pair( pp, qp );
                                        } // if
```

```
                                        } // for
                                    } // for
                                } // for
                            } // if
                        } // while

                        // Now, compute E from D
                        G.complement();
                        // And use it to compress the DFA.
                        compress( G );

                        assert( class_invariant() );
                        return( *this );
}
```

□

20 Hopcroft's algorithm

User function: *DFA::min_Hopcroft*

Files: `min-hop.cpp`

Uses: *CRSet, DFA, State, StateEqRel, StateSet*

Description: Member function *min_Hopcroft* implements Hopcroft's $n \log n$ minimization algorithm, as presented in [Wat93b, Algorithm 4.8].

Implementation: The member function uses some encoding tricks to effectively implement the abstract algorithm. The combination of the out-transitions of all of the *States* is stored in a *CRSet* C. Set L from the abstract algorithm is implemented as a mapping from *States* to **int** (an array of **int** is used). Array L should be interpreted as follows: if *State* q a representative, then the following pairs still require processing (are still in abstract set L):

$$([q], C_0), \ldots, ([q], C_{L(q)-1})$$

The remaining pairs do not require processing:

$$([q], C_{L(q)}), \ldots, ([q], C_{|C|-1})$$

This implementation facilitates quick scanning of L for the next valid *State-CharRange* pair.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:03 $
#include "crset.h"
#include "state.h"
#include "stateset.h"
#include "st-eqrel.h"
#include "dfa.h"

// Implement Algorithm 4.8 (Hopcroft's O(n log n) algorithm).
DFA& DFA::min_Hopcroft() {
	assert( class_invariant() );

	// This algorithm requires that the DFA not have any final unreachable
	// States.
	assert( Usefulf() );

	auto State q;

	// Keep track of the combination of all of the out labels of State's.
	auto CRSet C;
	for( q = 0; q < Q.size(); q++ ) {
		C.combine( T.out_labels( q ) );
	}

	// Encode set L as a mapping from State to [0,|C|] where:
	//	if q is a representative of a class in the partition P, then
	//	L (the abstract list) contains
	//		([q],C_0), ([q],C_1), ..., ([q],C_(L(q)-1))
	//	but not
	//		([q],C_(L(q))), ..., ([q],C_(|C|-1))
	auto int *const L( new int [Q.size()] );
	for( q = 0; q < Q.size(); q++ ) {
		L[q] = 0;
	}

	// Initialize P to be the total equivalence relation.
	auto StateEqRel P( Q.size() );
	// Now set P to be E_0.
```

```
	P.split( F );

	// Now, build the set of representatives and initialize L.
	auto StateSet repr( P.representatives() );
	if( F.size() <= (Q.size() - F.size()) ) {
		repr.intersection( F );
	} else {
		repr.remove( F );
	}

	// Do the final set up of L
	for( repr.iter_start( q ); !repr.iter_end( q ); repr.iter_next( q ) ) {
		L[q] = C.size();
	}

	// Use a break to get of this loop.
	while( 1 ) {
		// Find the first pair in L that still needs processing.
		for( q = 0; q < Q.size() && !L[q]; q++ );

		// It may be that we're at the end of the processing.
		if( q == Q.size() ) {
			break;
		} else {
			// Mark this element of L as processed.
			L[q]--;

			// Iterate over all eq. classes, and try to split them.
			auto State p;
			repr = P.representatives();
			for( repr.iter_start( p ); !repr.iter_end( p );
					repr.iter_next( p ) ) {
				// Now split [p] w.r.t. (q,C_(L[q]))
				auto State r( split( p, q, C.iterator( L[q] ), P ) );
				// r is the representative of the new split of the
				// eq. class that was represented by p.

				if( r != Invalid ) {
					// p and r are the new representatives.
					// Now update L with the smallest of
					// [p] and [r].
					if( P.equiv_class( p ).size()
							<= P.equiv_class( r ).size() ) {
						L[r] = L[p];
						L[p] = C.size();
					} else {
						L[r] = C.size();
					} // if
				} // if
			} // for
		} // if
	} // while

	// L is no longer needed.
	delete L;

	// We can now use P to compress the DFA.
	compress( P );

	assert( class_invariant() );
	return( *this );
}
```

□

21 A new minimization algorithm

User function: *DFA::min_Watson*

Files: `min-bww.cpp`

Uses: *CRSet, DFA, State, StateEqRel, SymRel*

Description: Member function *min_Watson* implements the new minimization algorithm appearing in [Wat93b, Sections 4.6–4.7]. The algorithm computes the equivalence relation E (on states) from below. The importance of this is explained in [Wat93b, p. 16].

Implementation: Function *min_Watson* is an implementation of Algorithm 4.10 (of [Wat93b]). The helper function *are_eq* is an implementation of the second algorithm appearing in Section 4.6 of [Wat93b]. Function *are_eq* takes two parameters more than the version appearing in [Wat93b]: H (a *StateEqRel*) and Z (a *SymRel*). These parameters are used to discover equivalence (or distinguishability) of *States* earlier than the abstract algorithm would.

Performance: Memoization in function *are_eq* would provide a great speedup.

```
/* (c) Copyright 1994 by Bruce W. Watson */
// $Revision: 1.1 $
// $Date: 1994/05/02 15:59:01 $
#include "state.h"
#include "crset.h"
#include "symrel.h"
#include "st-eqrel.h"
#include "dfa.h"
#include <assert.h>

// The following is function equiv from Section 4.6 of the min. taxonomy.
int DFA::are_eq( State p, State q, SymRel& S, const StateEqRel& H, const SymRel& Z ) const {
        if( S.contains_pair( p, q ) || H.equivalent( p, q ) ) {
                // p and q are definitely equivalent.
                return( 1 );
        } else if( !Z.contains_pair( p, q ) ) {
                assert( !S.contains_pair( p, q ) );
                assert( !H.equivalent( p, q ) );
                // p and q were already compared (in the past) and found to be
                // inequivalent.
                return( 0 );
        } else {
                assert( !H.equivalent( p, q ) );
                assert( Z.contains_pair( p, q ) );

                // Now figure out all of the valid out-transitions from p and q.
                auto CRSet C( T.out_labels( p ) );
                C.combine( T.out_labels( q ) );

                // Just make sure that p and q have the same out-transitions.
                auto int it;
                for( it = 0; !C.iter_end( it ); it++ ) {
                        if( (T.transition_on_range( p, C.iterator( it ) ) == Invalid )
                                        || (T.transition_on_range( q, C.iterator( it ) )
                                                == Invalid ) ) {
                                // There's something that one transitions on, but
                                // the other doesn't
                                return( 0 );
                        } // if
                } // for

                // p and q have out-transitions on equivalent labels.
                // Keep track of whether checking needs to continue.
                S.add_pair( p, q );
                for( it = 0; !C.iter_end( it ); it++ ) {
```

```
                        auto State pdest( T.transition_on_range( p, C.iterator( it ) ) );
                        auto State qdest( T.transition_on_range( q, C.iterator( it ) ) );
                        if( !are_eq( pdest, qdest, S, H, Z ) ) {
                                // p and q have been found distinguished.
                                S.remove_pair( p, q );
                                return( 0 );
                        }
                } // for

                // p and q have been found equivalent.
                S.remove_pair( p, q );
                return( 1 );
        } // if
}

DFA& DFA::min_Watson() {
        assert( class_invariant() );
        assert( Usefulf() );

        // (Symmetrical) State relation S is from p.14 of the min. taxonomy.
        auto SymRel S;
        S.set_domain( Q.size() );

        // H is used to accumulate equivalence relation E.
        auto StateEqRel H( Q.size() );
        // Start with the identity since this is approximation from below
        // w.r.t. refinement.
        H.identity();

        // Z is a SymRel containing pairs of States still to be considered.
        auto SymRel Z;
        Z.set_domain( Q.size() );
        Z.identity();
        // We will need the set of non-final States to initialize Z.
        auto StateSet nonfinal( F );
        nonfinal.complement();
        Z.add_pairs( F, nonfinal );
        // Z now contains those pairs that definitely do not need comparison.

        Z.complement();
        // Z initialized properly now.

        auto State p;
        // Consider each p.
        for( p = 0; p < Q.size(); p++ ) {
                auto State q;
                // Consider each q that p still needs to be compared to.
                for( Z.image( p ).iter_start( q ); !Z.image( p ).iter_end( q );
                                Z.image( p ).iter_next( q ) ) {
                        // Now compare p and q.
                        if( are_eq( p, q, S, H, Z ) ) {
                                // p and q are equivalent.
                                H.equivalize( p, q );
                        } else {
                                // Don't do anything since we aren't computing
                                // distinguishability explicitly.
                        }

                        // Comparing q to p is the same as comparing [q] and [p]
                        // all at once.
                        // Mark them as such.
                        Z.remove_pairs( H.equiv_class( p ), H.equiv_class( q ) );
                } // for
        } // for

        compress( H );
```

```
        assert( class_invariant() );
        return( *this );
}
```

□