Transduction Method for AIG Minimization

Yukio Miyasaka

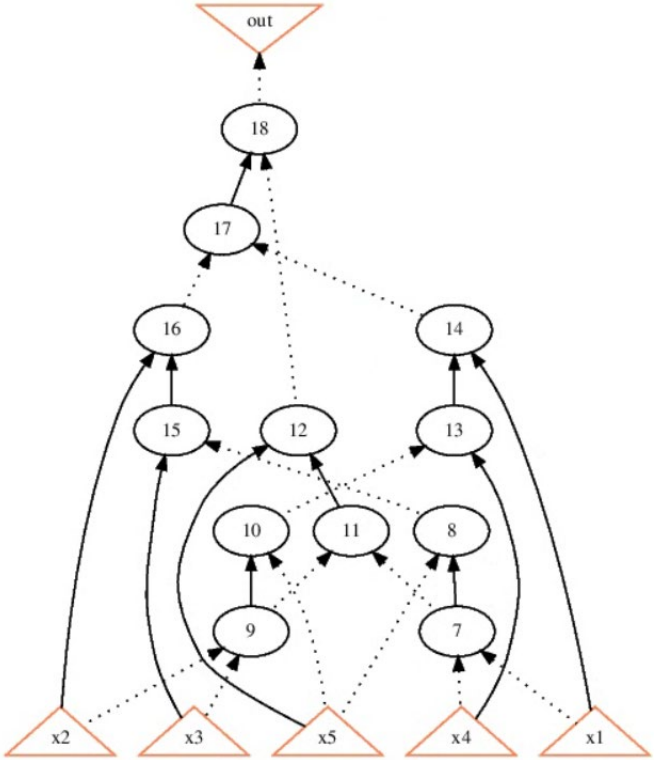
UC Berkeley

Introduction

- Increasing cost of transistors (1.5x increase between 2020 and 2022)
- It has become reasonable to spend more time in logic synthesis
- However, conventional methods focus on scalability rather than effectiveness
- In this work, we revisit the transduction method
  - It enumerates don’t-cares in the circuit
  - Computationally expensive but more powerful than conventional methods
- Our implementation **always derived smaller or equal-sized AIGs** for benchmarks of up to 1k nodes compared to the best results in the contest
  - Benchmarks were taken from IWLS 2022 Programming Contest
- Practical application will be optimization of highly reusable components such as multipliers

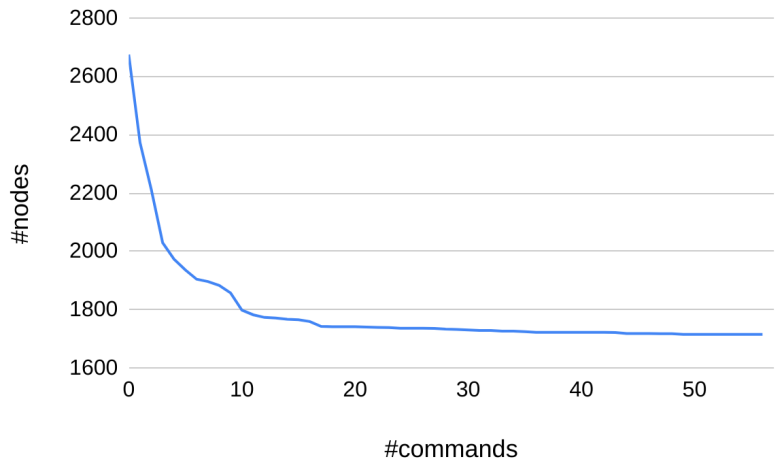
And-Inverter Graph (AIG)

- AIG is a directed, acyclic graph
- Each node has 2 inputs and corresponds to an AND gate
- Edges have an attribute that represents if there is an inverter
  - Dashed lines are inverters
- Mapping with other types of gates is performed in technology mapping afterwards
- We usually use AIG in logic synthesis
  - #nodes highly correlates with area, while #levels correlates with delay after mapping



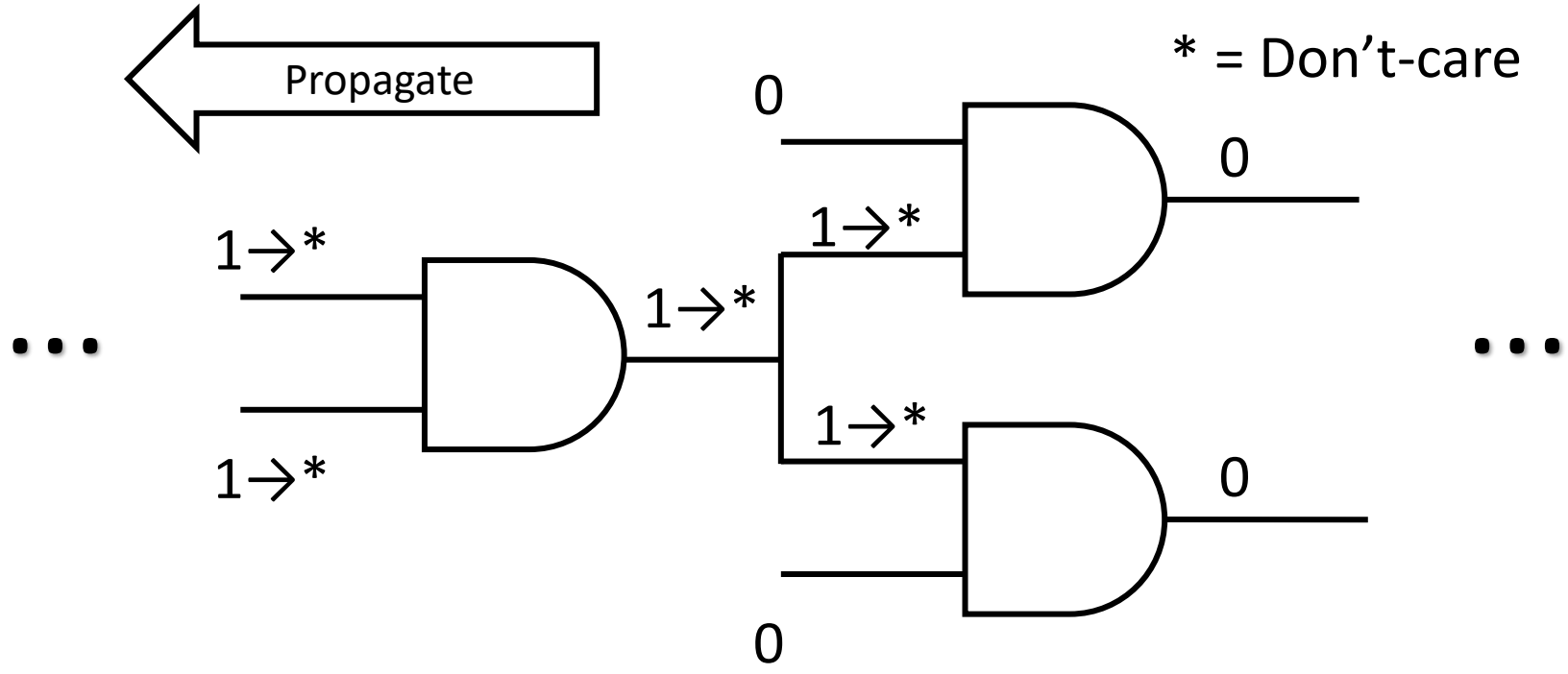
Background

- Modern logic synthesis is an incremental optimization process
- Initial circuits are derived from SOP, BDD, or bi-decomposition
- Various algorithms are run iteratively to improve the circuit quality
  - Balancing, rewriting, refactoring, resubstitution
- However, it often converges at a bad local minimum
- Explicit use of don’t-cares helps further optimize the circuit



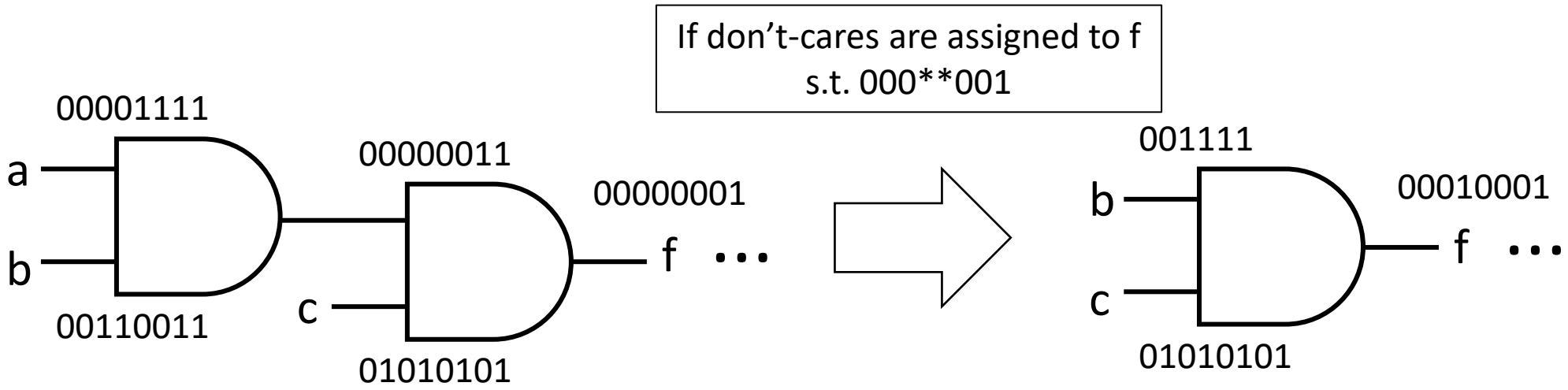
Don’t-Cares

- Don’t-cares arise from the controlling input of gates
  - If an input of AND gate is 0, the other inputs are don’t-care for the given input pattern
- Don’t-cares propagate to the fan-ins



Don’t-Care-Based Optimization

- With don’t-cares, we can find more optimization opportunities
- Rather than augmenting conventional algorithms with don’t-cares, we found it easier to implement don’t-care-based optimization separately

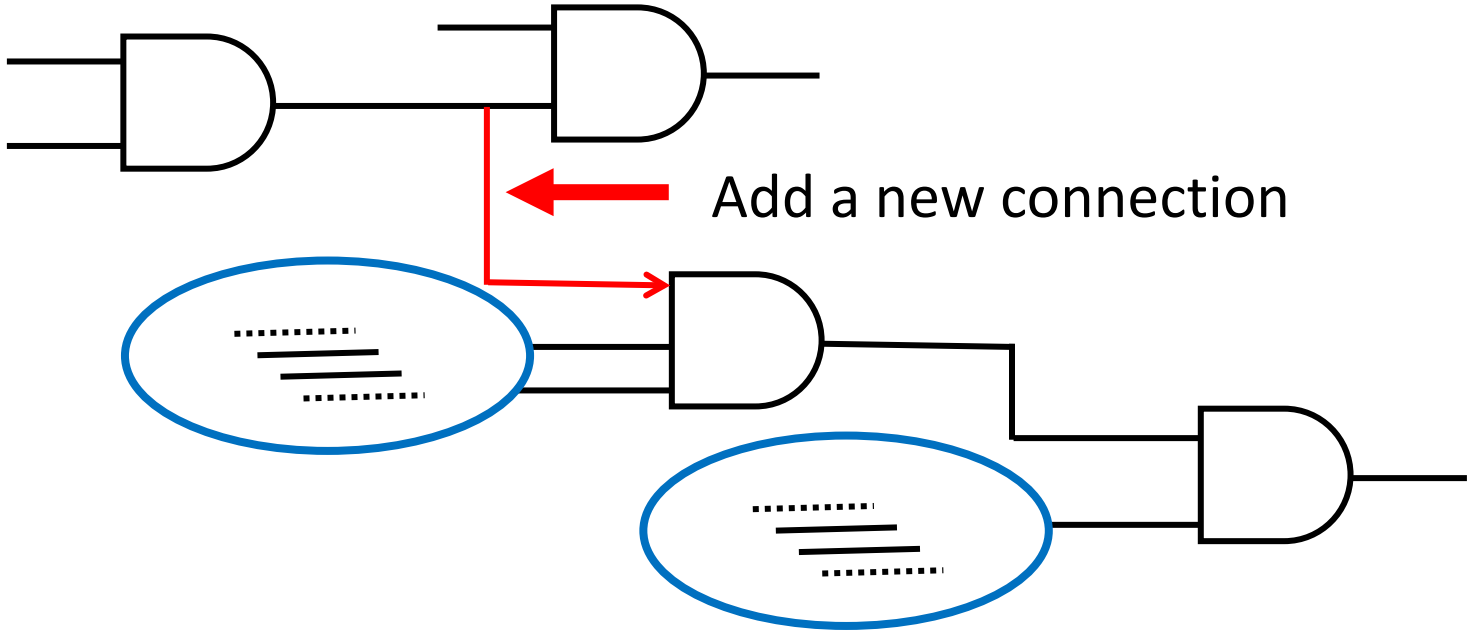


Transduction Method

- Transduction = Transformation + Reduction
- *Reduction* is also known as redundancy removal
  - It replaces wires that are always 1 [0] or don't-care with constant-1 [0, resp.]
  - It also removes redundant nodes that have no fanouts
- *Transformation* restructures the circuit using don't-cares
  - Not necessarily improves the circuit quality by itself
  - Transformation changes the distribution of don't-cares, and the subsequent Reduction might be able to remove some wires
  - There are various flavors of Transformation (substitution, merge, forwarding, resubstitution)
  - We implemented variants of resubstitution, which involves more heuristics than the others but is the most general

S. Muroga, Y. Kambayashi, H. C. Lai and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," in *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404-1424, Oct. 1989.

Overview of Transduction Method



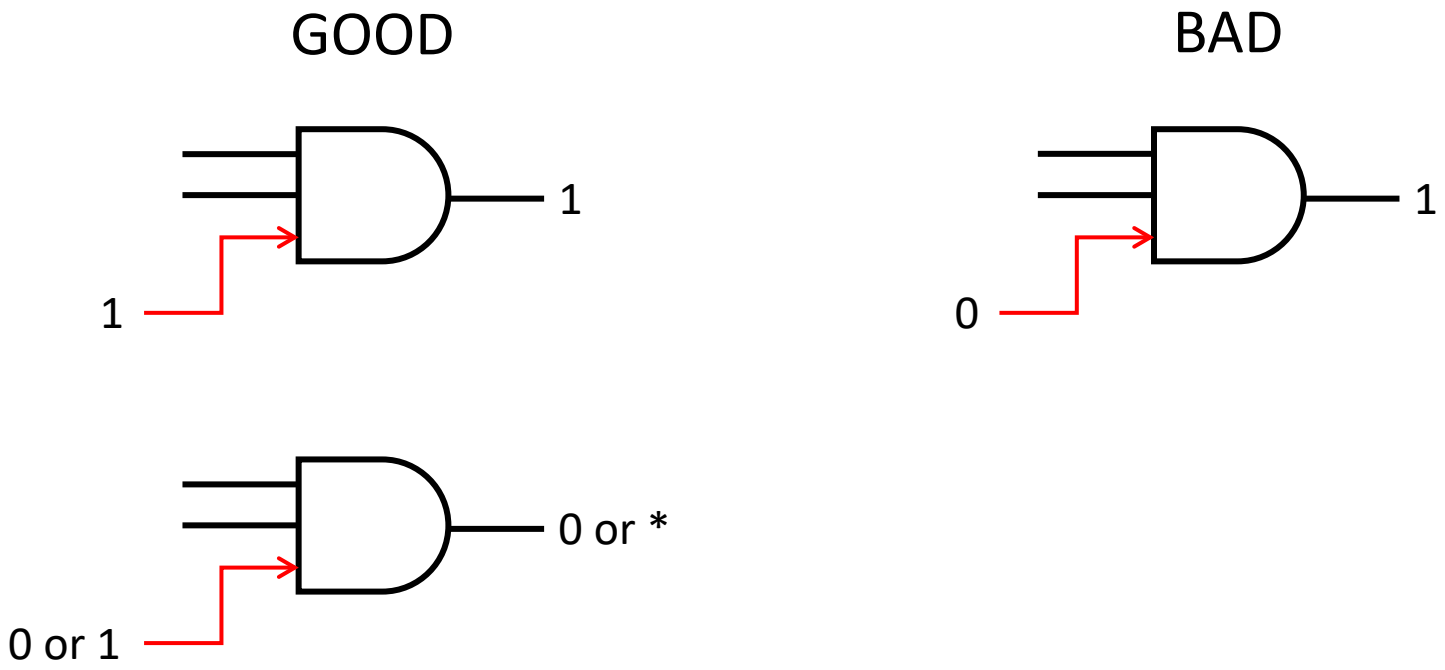
Some wires might become redundant

We extend AIG to MIAIG (multi-input AIG), where each node can take more than 2 inputs

- MIAIG can be converted into AIG simply by decomposing each node into 2-input nodes

Connectable Condition

- We add a connection that does not change the functions of primary outputs
- A wire is connectable if it never takes 0 when the node output is 1

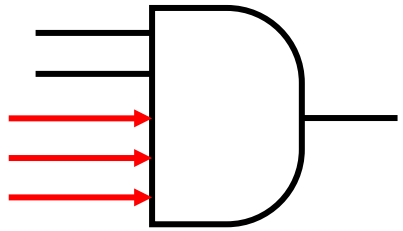


Proposed Transduction Method

- We focus on area reduction in terms of AIG nodes (2-input)
  - Delay (logic depth) or other measures could be optimized as well
- We propose three variants of resubstitution:
  - Resub
  - Resub-Mono
  - Resub-Shared

Resub

- Pick up a node
- Add all connectable wires to its fanin
- Apply Reduction
- If the area increases, undo

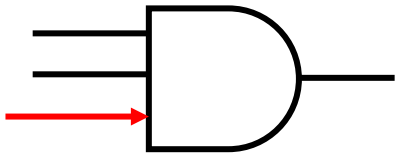


All possible fanins are added at the same time

Some heuristics

- Pick up a node in a reverse topological order
  - Worked well in our preliminary experiment
- After Reduction, the node is decomposed into 2-input nodes
  - This makes the intermediate functions available for subsequent Resub

Resub-Mono



Only one fanin is added each time

- Pick up a node
- For each connectable wire in a topological order:
  - Add the connectable wire to the fanin
  - Apply Reduction
  - If the area increases, undo

- Resub and Resub-Mono offset each other
  - Resub-Mono tries each connectable wire one by one (more exhaustive)
  - Resub covers the case that multiple wires work together to replace a fanin

Resub-Shared

- Globally restructures the circuit to create highly shared fanins
- This often increases #nodes, but the subsequent Resub/Resub-Mono may be able to reduce more than that
- Procedure:
  - Apply Resub while skipping both decomposition and undoing
    - Keep multi-input nodes and allow area growth
  - For each node in a topological order:
    - Find another node that shares fanins
    - If the fanins of A includes those of B, replace the shared fanins of A with the output of B
    - If they intersect but do not include each other, replace their shared fanins with a new node that takes the shared fanins as input (i.e., factoring)

Fanin Cost Function

- If multiple fanins are redundant at the same time, we need to decide which one to remove first
  - After removing one, others may be no longer redundant
- We use a heuristic cost function where one with the highest cost is removed first
- Cost is assigned as follows:

  Primary inputs < #fanouts (smaller is higher in cost) < one of the following

  1. Topological order
  2. #1's in the function (after negation if negated)
  3. #1's in the function (before negation)
  4. Pseudo random

Transduction Script (More heuristics!)

- We created a script based on preliminary experiments
  - Apply Resub-Mono first, and then Resub
    - Use compatible don’t-cares first, and then complete don’t-cares
  - Apply Resub-Shared when #nodes converges
  - Terminate if Resub and Resub-Mono end up in larger #nodes after Resub-Shared
- Parameters
  - Fanin cost function
  - Primary input order
  - Whether to apply Resub-Shared at the beginning
  - Compatible don’t-cares or complete don’t-cares in Resub-Shared
  - Whether to repeat Resub (Resub-Mono) in the innermost loop

Experiment

- Optimize AIGs synthesized from IWLS2022 benchmark
  - 4 starting points (“st”, “clp; st”, “clp; sop; fx; st”, “&ttopt”)
- We repeated the transduction script and ABC “dc2” and “if;mfs2”
  - Compared to the minimum results across all teams in the IWLS2022 contest
  - As an alternative baseline for small cases, we ran “&deepsyn”, which repeats those ABC commands, for the same amount of time as our method took
- Since it often gets stuck at local minimum, we restarted N times with different random seeds, depending on the size of AIG
  - Small (~70 nodes, 41 cases) N =100
  - Medium (~200 nodes, 23 cases) N = 10
  - Large (~1k nodes, 30 cases) N = 0 (with or without Resub-Shared)
    - For large cases, we only used the smallest starting point

Results (Small)

Benchmark	IWLS best	$N = 0$	$N = 100$	deepsyn	Time (minutes)
ex00	23	23	21	26	11
ex01	27	24	23	32	18
ex03	24	25	24	24	8
ex05	39	40	37	41	226
ex10	10	10	10	10	2
ex11	20	20	20	22	6
ex12	30	30	30	32	24
ex13	46	47	42	45	148
ex14	63	60	58	65	874
ex16	18	18	18	18	4
ex17	24	24	24	24	9
ex18	32	32	32	33	30
ex19	38	38	38	46	93
ex20	56	53	49	52	253
ex21	70	62	56	57	729
ex28	39	39	39	39	41
ex29	39	37	35	39	63
ex32	44	44	44	46	47
ex34	46	48	45	47	97
ex35	15	15	15	17	2
ex38	28	28	27	31	28
ex41	17	17	17	17	3
ex42	28	28	28	28	10
ex43	37	37	37	37	37
ex44	58	53	49	63	637
ex46	32	32	31	32	24
ex47	25	25	25	25	4
ex49	39	39	39	39	37
ex50	18	18	18	18	3
ex51	29	26	26	30	24
ex52	19	19	18	19	5
ex53	40	36	34	41	70
ex54	12	12	12	13	3
ex56	29	29	29	29	4
ex60	67	60	53	66	888
ex62	40	40	40	40	8
ex92	29	29	29	32	15
ex93	41	41	39	45	84
ex94	36	34	33	41	82
ex95	65	61	58	67	753
ex97	69	61	55	69	800
Total	1461	1414	1357	1497	6205

Green (win), Red (loss), Blue (tie)

Compared to IWLS best:

N=0	12 wins	4 losses	25 ties
N=100	20 wins	no losses	21 ties

Compared to deepsyn:

N=0	22 wins	5 losses	14 ties
N=100	28 wins	no losses	13 ties

Results (Summary)

Total number of AIG nodes

	IWLS best	Proposed
Small	1461	1357
Medium	3107	2495
Large	11150	9206

- For medium and large cases, our method never lost against IWLS best
  - Medium: 22 wins and 1 tie
  - Large: 29 wins and 1 tie
- Regarding runtime,
  - Small cases took less than a day for 100 restarts with 4 starting points
  - A half of medium and large cases finished within a day
  - The other half required very high runtime, a week on average, but less than a month

Conclusion

- We proposed variants of transduction method for AIG minimization
- Our script interleaves transduction methods and ABC commands, and generated smaller or equal-sized AIGs for all cases up to 1k nodes compared to the best results of the IWLS 2022 contest
  - It also never lost against pure ABC commands using fair runtime
  - Transduction methods are crucial for improving lower bound and robustness
- High runtime may be compensated by parallelizing independent runs and adopting a heuristic search algorithm
  - Backend may be changed from BDD to SAT or truth table
- Xor-AIG is a possible extension

Implementation Detail

Compatible set and maximum (complete) set of don’t-cares

- Compatible set is a subset of don’t-cares where each don’t-care is independent
  - A new connection affects only the fanin cone
- Maximum set is a full set of don’t-cares, needs more recalculations
  - Whenever a wire is removed, we have to recalculate the fanout cone
  - When a gate has different paths to reach another gate in the fanout cone, we have to simulate the fanout cone with a complemented output and calculate a miter with the original circuit

BDD

- We use BDDs to represent the functions and don’t-cares in the circuit
- Two BDDs for each wire, one for function and the other for don’t-care

Medium

Tie in one case

For ex27, N=10 gives 119 node AIG, much smaller than IWLS best

Benchmark	IWLS best	$N = 0$	$N = 10$	Time (hours)
ex02	88	79	77	5
ex15	82	79	73	9
ex22	86	72	69	5
ex23	104	109	87	12
ex24	116	94	88	25
ex25	146	112	94	49
ex26	163	153	99	111
ex27	183	188	119	361
ex30	68	68	68	85
ex33	77	74	71	3
ex37	147	146	146	207
ex39	202	174	163	396
ex40	195	184	178	392
ex45	191	185	183	493
ex55	156	120	112	58
ex58	92	75	73	7
ex73	208	113	93	476
ex84	134	115	115	55
ex85	202	178	171	477
ex86	165	148	144	252
ex96	76	69	67	3
ex98	142	135	129	149
ex99	84	79	76	4
Total	3107	2749	2495	3634

Large

Tie in one case

For these large benchmarks, Resub-Shared sometimes gives us worse results
(It also takes more time because we get larger intermediate AIGs after Resub-Shared)

Benchmark	IWLS best	$N = 0$ w/ RS	$N = 0$ w/o RS	Min	Time (hours)
ex04	304	288	288	288	17
ex07	166	141	174	141	3
ex08	544	550	513	513	238
ex09	555	536	515	515	231
ex48	470	482	462	462	360
ex57	180	116	200	116	2
ex59	277	242	253	242	27
ex63	801	285	529	285	666
ex64	442	391	373	373	79
ex66	329	334	329	329	18
ex68	266	211	204	204	6
ex69	245	170	166	166	6
ex70	225	176	206	176	5
ex71	324	242	271	242	18
ex72	369	216	355	216	84
ex74	437	194	319	194	64
ex75	477	367	368	367	33
ex76	240	214	230	214	8
ex77	277	253	256	253	20
ex78	339	293	313	293	40
ex79	332	284	332	284	34
ex80	497	494	483	483	178
ex81	326	304	319	304	43
ex82	572	535	523	523	238
ex83	588	549	551	549	446
ex87	372	355	351	351	36
ex88	295	284	287	284	13
ex89	212	186	193	186	7
ex90	449	446	424	424	76
ex91	240	229	232	229	4
Total	11150	9367	10019	9206	3002