On Sub-optimality and Scalability of Logic Synthesis Tools

Igor L. Markov and Jarrod A. Roy

Department of EECS, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122
{imarkov, royj}@eecs.umich.edu

Abstract

The development of EDA tools is driven by the desire to find near-optimal solutions for circuits of increasing size. However, quantifying sub-optimality and scalability of optimization heuristics is non-trivial. We follow related developments in physical design where tools for circuit partitioning and placement miss optimal solutions by up to a factor of two on multi-million-gate netlists. We estimate the growth of optimal costs in specific scalable logic synthesis problems and show empirically that existing tools `Espresso`, `SIS` and `BDS` appear to produce exponentially sub-optimal circuits in several cases. While small circuits for multiplication are well-known and are "instantiated" by commercial tools rather than synthesized from scratch, we demonstrate a logic synthesis problem where all known answers appear exponentially sub-optimal.

1 Introduction

Problems solved by modern EDA tools are often computationally intractable from the worst-case perspective, and algorithm developers have to resort to heuristics. Empirical evaluation is typically performed on industrial circuits where optimal solutions are unknown. Thus, it is difficult to say how much improvement can be achieved by algorithm innovation in the future — a question of interest to EDA vendors, researchers and funding agencies.

Several works in physical design propose to evaluate the optimality of existing EDA tools by constructing artificial benchmarks with known solutions [HagenHK95, ChangCX03] or upper bounds on optimal costs [HagenHK95, CongRX03]. As a result, sub-optimality gaps ranging from 40% to over 100% have been demonstrated for recent VLSI placers in terms of half-perimeter wirelength [ChangCX03]. Since artificial benchmarks are not limited by size and can be scaled up astronomically, it was shown that currently-achievable placements of larger circuits (up to 2 million gates) exhibit growing sub-optimality, but some placers scale reasonably well. In hypergraph partitioning, similar techniques are used in [CongRX03] to compare two min-cut bisectors, of which one consistently tracks optimal costs while the other sometimes misses by up to 30% and is asymptotically slower. VLSI routing instances with known good solutions were constructed in [AloulRMS03] using a technique called "flooding". Needless to say, good performance on artificial benchmarks does not guarantee good performance on industrial circuits. However, improving the performance of existing tools on artificial benchmarks may lead to improvements in the real-world, and the simpler structure of artificial benchmarks may facilitate more efficient algorithms engineering, as well as program debugging.

Our work addresses similar questions for logic synthesis tools. It includes both theoretical analyses and empirical results produced with existing tools Espresso, SIS and BDS. Moreover, we propose a scalable synthesis problem for which all of those tools appear to produce exponentially sub-optimal circuits. Unlike for the synthesis of multiplier circuits, no polynomially-sized circuits are currently known, despite an abstract proof that such circuits exist. Because of this, synthesis tools cannot store pre-computed circuits and instantiate them on demand, as is done with multipliers. Thus, our experiments address the limits of scalability for existing logic-synthesis tools and cast doubts on folklore claims that "RTL-to-GDSII design automation is a solved problem". We also point out that, unlike in physical design, where demonstrated sub-optimality is largely limited by constant factors, existing logic synthesis tools apparently produce exponentially sub-optimal solutions.

The remaining part of this manuscript is organized as follows. Previous work is surveyed in Section 2. In Section 3, we examine the performance of synthesis tools on common synthesis problems. In Section 4, we propose a way to evaluate the sub-optimality and scalability of synthesis tools and show related empirical results. Conclusions and future work are summarized in Section 5.

2 Previous Work

Unlike in logic synthesis, many layout problems are geometric in nature and easy to visualize. This often enables straightforward constructions, such as grid circuits, with obvious optimal layouts. A 10×10 grid is illustrated in Figure 5, where a unique optimal placement is shown.[1] Grid-graphs have been used to debug partitioners and placers for at least 20 years — they are convenient as testcases because sub-optimalities can be visualized and traced to relevant algorithms and implementations [AdyaEtAl03]. Appendix A describes more sophisticated ideas proposed to study the sub-optimality and scalability of physical design tools.

3 Common Logic Synthesis Problems

In this section, we gauge the performance of existing logic synthesis tools using three common tasks: parity, addition, and multiplication. All our empirical results involve three publicly available logic synthesis tools — Espresso [McGeerSBS93, UCBTech], SIS [SentovichEtAl92, UCBTech] and BDS [YangC02]. Espresso is a two-level logic simplification tool, which we use in the "exact" mode so that the returned circuits are provably the smallest possible. SIS is a software package for multi-level logic optimization. It does not guarantee achieving minimum gate counts, and one might hope that it would proceed significantly beyond Espresso's limitations. BDS is a multi-level logic minimization tool that is based on Binary Decision Diagrams, which makes it very different from SIS and Espresso. We use truth tables as inputs to these tools and tabulate resulting reductions in circuit size. Gate counts are plotted on a *log-log* scale where polynomial functions tend to straight lines and non-linear trends are indicative of exponential asymptotic behavior.

3.1 Parity Testing

Determining the parity of n input bits is a fairly simplistic problem — a circuit comprised of $n-1$ XOR gates can solve it. Surprisingly, neither Espresso nor SIS were able to reduce such circuits at all. However, BDS found the solution using $n-1$ XOR gates.

3.2 Addition

Adders of varying sizes are common in many applications, so implementing them in small circuits is a practical task.

Small and simple addition circuits are well-known. Ripple-carry adders add two n-bit integers in $\Theta(n)$ time and have $\Theta(n)$ size [CLR90]. Carry-lookahead adders take $\Theta(\lg n)$ time and require $\Theta(n \lg n)$ space [Hennessy96]. Both carry-skip and carry-select adders take $\Theta(\sqrt{n})$ time and require $\Theta(n)$ space [Hennessy96].

Espresso was able to reduce up to 7-bit adders, and SIS could reduce up to 8-bit adders. Figure 1 illustrates the growth of gate counts in addition circuits produced by Espresso and SIS, on a *log-log* scale.

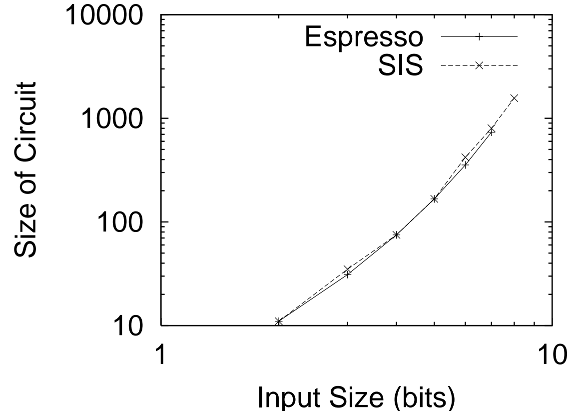

Figure 1: Gate counts for adders returned by Espresso and SIS. The near-linear trend in the *log-log* plot suggests near-polynomial growth in circuit size.

3.3 Multiplication

Automatic synthesis of multiplication circuits is more difficult than synthesis of circuits for parity-testing and addition. However, this task is very significant in applications.

Simple and small multiplication circuits are well-known. Array multipliers multiply two n-bit integers in $\Theta(n)$ time and have $\Theta(n^2)$ size [CLR90]. Wallace-tree multipliers take $\Theta(\lg n)$ time and take up $\Theta(n^2)$ space [CLR90].

In our experiments, Espresso was able to reduce up to 4-bit multipliers. Figure 2 shows the growth of the multiplier circuits produced by Espresso and SIS. Because there are only 3 data points, it is difficult to make any judgments on the trend of the size multiplication circuits produced by Espresso. SIS, on the other hand, appears to produce multiplication circuits that are super-polynomially sized. BDS could only complete the trivial 2-bit multiplication test.

The fact that automatic synthesis of multication circuits defies many synthesis tools is well-known. As multiplication circuits have been extensively studied, many commercial tools contain tables of multipliers with varying characteristics and "instantiate" such circuits rather than synthesize them from scratch. However, such an approach may fail for other difficult but less-studied synthesis problems.

[1] A simple induction argument proves that if all edges have Manhattan length one, then the placement is uniquely determined. On the other hand, none of the edges can be shorter.

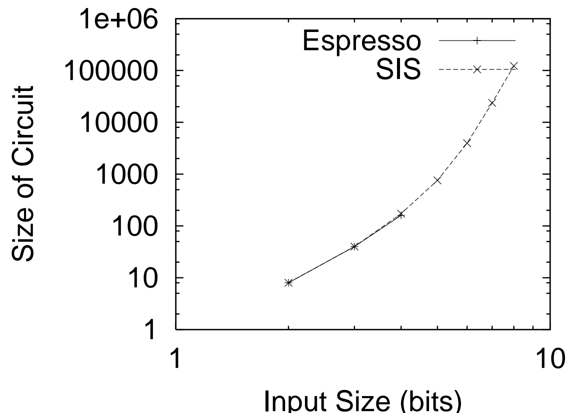

Figure 2: Sizes of multiplication circuits produced by Espresso and SIS. The non-linear trend in the *log-log* plot suggests super-polynomial growth of circuit size for SIS.

4 Primality Testing as a Problem in Logic Synthesis

On August 6, 2002, Manindra Agrawal, Neeraj Kayal and Nitin Saxena presented a deterministic polynomial time algorithm for primality testing [AKS02]. Despite being discovered by undergraduate students and their advisor, this result has been a major breakthrough in primality testing and in a week made it to the New York Times [Robinson02]. While its immediate applications, e.g., in cryptography, currently appear insignificant, this result implicitly guarantees the existence of polynomial-size circuits for primality testing. Finding such circuits explicitly appears difficult, and we propose to evaluate the scalability and sub-optimality of modern logic synthesis tools by trying to synthesize these circuits. We begin by estimating the asymptotic growth of best-possible primality-testing circuits, based on results in [AKS02].

4.1 Loose Upper Bounds

A well-known result in Complexity Theory [Sipser97] is that if a deterministic single-tape Turing Machine runs in time $p(m)$, where m is the size of the input to the machine, then there is an equivalent combinational circuit of size $(p(m))^2$. Thanks to the deterministic polynomial-time algorithm in [AKS02], poly-sized primality testing circuits must exist. The AKS algorithm runs in $O(n^{12} * \text{poly}(\log n))$ time on n-bit integers [AKS02]. This result assumes a RAM architecture which is appropriate for computers but not Turing Machines. To get an upper bound on the size of the combinational circuits in question, a time bound for Turing Machines must be derived.

For a description of the AKS algorithm, see Appendix B. The first two main sections of the AKS algorithm use a constant number of temporary integers of size at most n bits. Thus these sections of the algorithm can be performed by a multiple-tape Turing Machine in the same amount of asymptotic time as the RAM machine.

Single-tape Turing Machines can simulate multiple-tape Turing Machines and, in fact, have the same computational power [Sipser97]. In one of the simulation techniques, the single-tape machine combines all the data from the multiple-tape machine onto its tape with special markers for the head positions of the multiple-tape machine. For every step of the multiple-tape machine, the single-tape machine must scan its entire tape and thus incurs a time penalty proportional to the overall amount of data. Since the amount of data on all of the tapes is proportional to n, the first part of the algorithm will take $O(n^4)$ time on a Turing Machine and the second part will take $O(n^{10} * \text{poly}(\log n))$ time.

The last part of the AKS algorithm takes the bulk of the time and uses much more memory than the two previous parts of the algorithm. The last part of the algorithm consists of a loop that cycles $O(nr^{\frac{1}{2}})$ times. In each cycle of the loop, a polynomial of degree less than r is raised to the power of the input integer. Using the repeated squaring method there are $O(n)$ polynomial multiplications per polynomial exponentiation. Using the "grade school" polynomial multiplication algorithm, one multiplication can be done using $O(r^2)$ integer multiplications and additions of n-bit integers. Multiplication of n-bit integers can be done in $O(n^2)$ time and additions can be done in $O(n)$ time. Using these naive methods, the last part of the AKS algorithm will take $O(n^4 r^{\frac{5}{2}})$ time. These naive techniques can be implemented on a multiple-tape Turing Machine with a constant number of tapes. Converting to a single-tape machine, the scanning penalty for each step of the multiple-tape machine is $O(nr)$ since that is the amount of memory needed to store the polynomials. Thus the final time for a single-tape machine comes down to $O(n^5 r^{\frac{7}{2}})$. In [AKS02] an upper bound of $O(n^6)$ was proven for r, so an upper bound for the last part of the AKS algorithm (and thus the entire algorithm) on a single-tape Turing Machine is $O(n^{26})$. Thus an upper bound on circuit size is $O(n^{52})$. If the Sophie Germain Conjecture [AKS02] is true, the AKS time bound reduces dramatically to $O(n^{12})$ and the circuit size bound to $O(n^{24})$.

4.2 Empirical Results

In this series of tests, each tool reads a truth-table for an n-bit primality tester, i.e., a complete list of n-bit primes, and is asked to return a smallest possible circuit implementation. The input is exponential as the number of n-bit primes is $\Theta\left(\frac{2^n}{n}\right)$, and thus automatically producing polynomial-sized primality tester circuits requires non-trivial hardware resources. We used workstations with 2Gb of RAM and 2.0GHz Pentium4-Xeon processors.

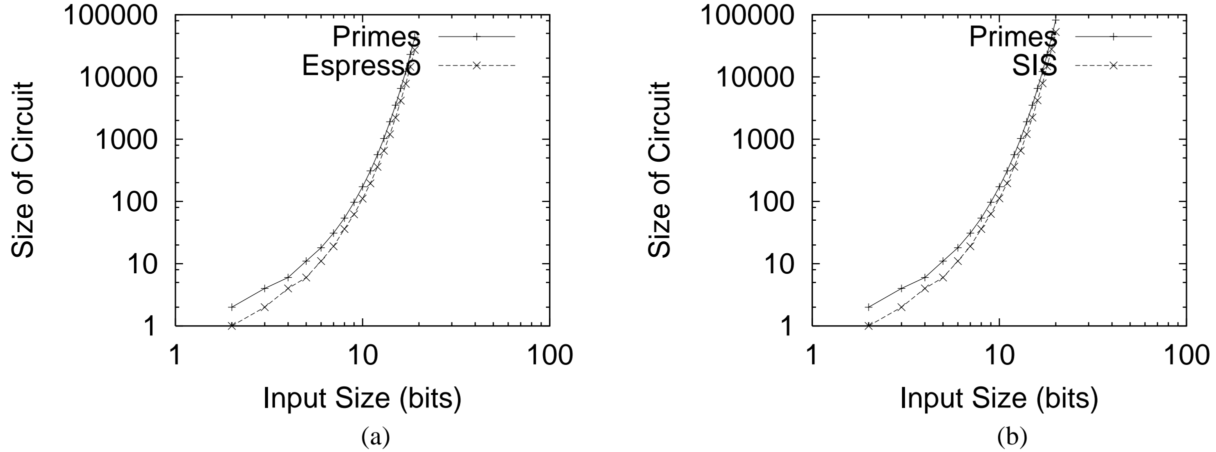

Figure 3: Sizes of circuits for primality testing returned by (a) Espresso, and (b) SIS. The "primes" line represents exponentially growing input circuits which implement the primality-testing truth table. Non-linear trends for Espresso and SIS in these *log-log* plots suggest exponential growth of circuit sizes.

Figure 3 (a) illustrates the performance of Espresso. We first plot, on a *log-log* scale, the number of n-bit primes against n, which is directly related to the size of the input circuit given to Espresso. We also plot the size of the circuits returned by Espresso for up to 19-bit primes (it failed on a 20-bit primality tester). Figure 3 (a) suggests that the size of the circuits produced by Espresso grows super-polynomially. This leaves at least least two possibilities: (i) two-level minimization is not powerful enough for primality testing, or (ii) the circuits will eventually become polynomially-sized for input sizes much greater than 19 bits. In the first case, one might try tools for multi-level logic optimization.

Unfortunately, SIS was only able to minimize up to 20-bit primality testing circuits. We ran SIS with commands `full_simplify` and script `rugged` and plot the best result achieved in Figure 3 (b). These empirical data suggest that SIS gives about the same results as Espresso, so the original possibilities still stand. Finally, we tried a more recent package — BDS.

Unfortunately, as Figure 4 shows, BDS is only able to minimize primality testing circuits with up to 7-bits input. The circuits produced by BDS are no better than those produced by Espresso and SIS.

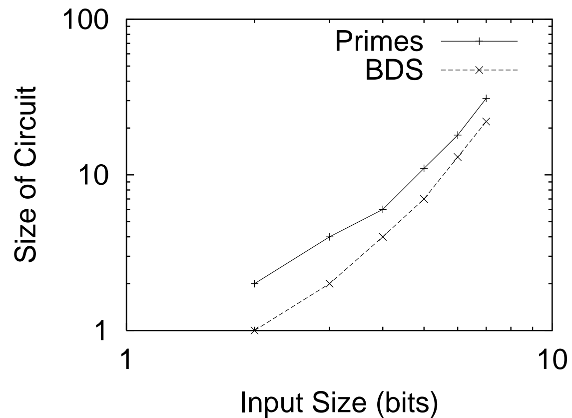

Figure 4: Sizes of circuits for primality testing returned by BDS. Primes represents the exponentially growing circuit size given to BDS. The non-linear trend for BDS in this *log-log* plot suggests exponential growth of circuit size.

5 Conclusions and Further Work

The main result of our work is the identification of a somewhat extreme circuit synthesis task for which the best known circuits (including those produced by existing tools) appear exponentially sub-optimal. While the synthesis of adders and multipliers is known as a difficult problem, primality testing appears even harder because good primality testers are unknown, while good adders and multipliers can be found in textbooks. In fact, new experimental synthesis tools not evaluated in our paper, e.g., M31, manage to automatically find good adders and multipliers.

We hope that our work will generate further research and lead to improvements in synthesis software and the theory of logic circuits. Further work beyond primality-testing includes scalability studies based on doubling constructions in the spirit of [HagenHK95]. For example, one can join by an AND-gate two copies of the same circuit applied to disjoint sets of inputs, and this will produce an upper bound for optimal circuits implementing the new function with twice the inputs.

Much more can be done along the lines of primality testing. From the Espresso results, it appears that two-level logic minimization is not a powerful enough technique to find the polynomially sized circuits that are guaranteed to exist. Otherwise, our empirical data are somewhat inconclusive — with only 19 datapoints, it is difficult to argue about polynomials of degree 24 or more. Indeed, a rough upper bound on circuit size is $O(n^{52})$, and 54 or more data points would be needed. However, the existing 19-20 datapoints seem to suggest exponential growth rather than a sophisticated polynomial. At this point, it is still worthwhile to try various logic minimization software to collect more datapoints. Scalability may be improved by specialized synthesis methods and better input encoding.

We only analyzed fairly simple primality-testing algorithms because they can be easily implemented on a Turing Machine. Using Fast Fourier multiplication would be far more complicated, but could improve upper bounds.

When converting Turing Machines to combinational circuits, a much tighter bound on circuit size is available if the Turing Machine is *oblivious*. Implementing AKS on an oblivious machine may lead to better overall bounds for combinational primality testers. Finally, one should track on-going improvements over the original AKS algorithms.

References

[AdyaEtAl03] S. N. Adya et al., "Benchmarking for Large-scale Placement," *ISPD 2003*, pp. 95-103.

[AloulRMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. S. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presense of Symmetry," to appear in *IEEE Trans. on CAD* 2003 (earlier version in *DAC 2002*, pp. 731-736).

[AKS02] M. Agrawal, N. Kayal, and N. Saxena, "PRIMES is in P," preprint, August 2002. `http://www.cse.iitk.ac.in/news/primality.html`

[BOOKSHELF] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "The GSRC Bookshelf for Fundamental CAD Algorithms," `http://vlsicad.eecs.umich.edu`.

[ChangCX03] C. C. Chang, J. Cong and M. Xie, "Optimality and Scalability Study of Existing Placement Algorithms," *ASP DAC* 2003, pp. 621-627.

[CLR90] T. H. Cormen, C. H. Leiserson and R. L. Rivest, "Introduction to Algorithms," *The MIT Press/McGraw-Hill Book Company* , 1990, pp. 660-677.

[CongRX03] J. Cong, M. Romesis, and M. Xie, "Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms", *ISPD* '03, pp. 88-94.

[HagenHK95] L. Hagen, J. H. Huang, and A. B. Kahng, "Quantified Suboptimality of VLSI Layout Heuristics", *DAC* 1995, pp. 216-221.

[Hennessy96] J. L. Hennessy and D. A. Patterson, "Computer Architecture A Quantitative Approach," *Morgan Kaufmann* , 2nd edition, 1996, p. A-46.

[McGeerSBS93] P. McGeer, J. Sanghavi, R. K. Brayton and A. Sangiovanni-Vincentelli, "ESPRESSO-Signature: A New Exact Minimizer for Logic Functions," *IEEE Trans. on VLSI*, vol. 1 no. 4, December 1993, pp. 432-440.

[Robinson02] S. Robinson, "New Method Said to Solve Key Problem in Math," The New York Times Late Edition - Final, Section A, p. 20, column 1, August 8, 2002. `http://www.nytimes.com/2002/08/08/science/08MATH.html`

[SentovichEtAl92] E. M. Sentovich et al., "SIS: A System for Sequential Circuit Synthesis", `http://citeseer.nj.nec.com/sentovich92sis.html`

[Sipser97] M. Sipser, "Introduction to the Theory of Computation," *PWS Publishing Co.*, 1997.

[UCBTech] UC Berkeley Technology Warehouse, `http://www-cad.eecs.berkeley.edu/Software/software.html`

[YangC02] C. Yang and M. Ciesielski, "BDS: A BDD-based Logic Optimization System," *IEEE Transactions on CAD*, Vol. 21, July 2002, pp. 866-876. `http://www.ecs.umass.edu/ece/labs/vlsicad/bds/bds.html`

A Sub-optimality in Physical Design

Construction of realistic irregular graphs with "built-in" optimal placements is credited to Boese in [HagenHK95, Section 1]:

> Given a netlist hypergraph $G_H = (V, E)$ and an array of $|V|$ placement slots, the idea is to construct a new hypergraph G'_H which optimally assigns terminals of each hyperedge onto a number of contiguous slots equal to the hyperedge size. The resulting G'_H can be "difficult" to distinguish from G_H, yet the optimum placement (Manhattan wirelength) of G'_H is known.

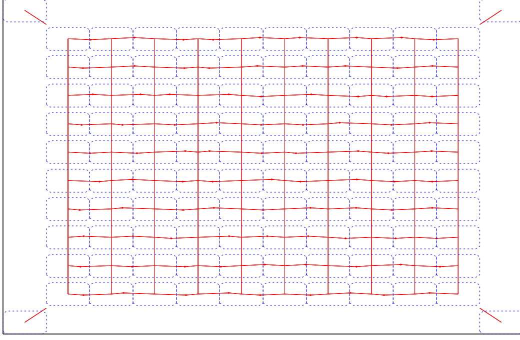

Figure 5: An optimally placed 10×10 grid-graph ($n \times n$) with 104 vertices ($n^2 + 4$) and 184 edges ($2n^2 - 2n + 4$). The four vertices at the corners are fixed.

```
Input:  integer n > 1

 1. if ( n is of the form a^b, b > 1 ) output COMPOSITE;
 2. r = 2;
 3. while(r < n) {
 4.        if ( gcd(n,r) ≠ 1 ) output COMPOSITE;
 5.        if (r is prime)
 6.              let q be the largest prime factor of r − 1;
 7.              if (q ≥ 4√r log n) and (n^((r−1)/q) ≢ 1 (mod r))
 8.                    break;
 9.        r ← r + 1;
10. }
11. for a = 1 to 2√r log n
12.        if ( (x − a)^n ≢ (x^n − a) (mod x^r − 1, n)) output COMPOSITE;
13. output PRIME;
```

Figure 6: The AKS algorithm (pseudocode from [AKS02]).

As in the $n \times n$ grid example, every hyperedge independently achieves the smallest possible wirelength, therefore the placement is optimal. However, multiple optimal placements may now exist. Specific "Placement Examples with Known Optima" (PEKO) have been created along these lines recently [ChangCX03] and match the net-degree distribution of well-known ISPD 98 circuit benchmarks released by IBM. The PEKO benchmarks, as well as grids can be downloaded through the GSRC Bookshelf [BOOKSHELF]. None of existing placers produce optimal solutions on PEKO benchmarks — the sub-optimality ratio ranges approximately from 1.4 to 2.0. Several placers are able to place small grids optimally, but not larger grids [AdyaEtAl03]. Moreover, stochastic search algorithms, such as Simulated Annealing, have difficulties finding unique optimal placements of grids, and this may also apply to regular datapath-like circuits.

The notion of scalability is studied in particular detail in [HagenHK95], where the authors argue that *knowing optimal costs is not necessary to evaluate how well a heuristic scales in terms of solution quality*. They propose a construction in which multiple copies of a netlist are connected in such a way that the optimal wirelength or the optimal cut of the new netlist grow linearly with the number of vertices. For a given heuristic, one then tabulates the rate of growth of solution costs. The asymptotic sub-optimality of the heuristic is measured by how super-linear that rate is.

The authors of [CongRX03] propose benchmarks for hypergraph partitioning with built-in upper bounds rather than known optimal costs. Difficult global routing instances with known good, but not necessarily optimal, solutions have been proposed in [AloulRMS03].

B The AKS Algorithm

The AKS algorithm is based on the following theorem for prime numbers [AKS02]:

Theorem. *Suppose that a is coprime to p. Then p is prime if and only if*

$$(x-a)^p \equiv (x^p - a)(\text{mod } p)$$

Testing this identity explicitly takes time linear in p since there are $p+1$ coefficients binomial on the left hand side. Since the magnitude of p is exponential in the number of bits, this explicit approach is infeasible. Instead, the AKS algorithm evaluates the identity modulo a polynomial of the form $x^r - 1$. The algorithm first chooses a "suitable" value for r and then repeatedly evaluates the following condition for a "small" number of a's [AKS02]:

$$(x-a)^p \equiv (x^p - a)(\text{mod } x^r - 1, p)$$

We reproduce the original pseudocode for the AKS algorithm [AKS02] in Figure 6. It shows how a suitable value for r is chosen and which values of a are tested. Detailed correctness proofs and complexity analyses can be found in [AKS02].