SATISFIABILITY

AND

THE ART OF COMPUTER PROGRAMMING

DON KNUTH — SAT 2012 — TRENTO

THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 6A

A (VERY INCOMPLETE) DRAFT OF SECTION 7.2.2.2: SATISFIABILITY

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

http://www-cs-faculty.stanford.edu/~knuth/
fasc6a.ps.gz

* * *

Zeroth printing (revision −93), 10 June 2012

* * *

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as ...

At the moment I'm just beginning to write this material, so it's in an *extremely* raw state. But hey, I had to start somewhere. As I finish drafting small pieces of the final big picture, I'm trying them out by making these test pages. You will soon see, however, that I've got a *long* way to go before I'll have anything coherent. These are scraps that I hope to refine and polish (if the FORCE stays with me). You'll also notice that many of the equation numbers and exercise numbers are flaky, because I keep changing them frequently.

* * *

My notes on combinatorial algorithms have been accumulating for more than fifty years, yet I fear that in many respects my knowledge is woefully behind the times. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 90, 91, 108, . . . ; I've also implicitly mentioned or posed additional

unsolved questions in the answers to exercises 64, 80, 81, 90, 108, 185, 300, 323, ... Are those problems still open? Please inform me if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises 1, 2, 3, 4, 14, 61, 62, 68, 80(b,c,d), 82, 83, 93, 160, 168, 178, 179, 180, 235, 236, 237, 240, 251(b), 252, 280, 293, 320, 321, 324, ..., and/or the answers to exercises ....

Special thanks are due to Armin Biere, Niklas Eén, Svante Janson, Oliver Kullmann, Wes Pegden, Niklas Sörensson, and ... for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections.

I happily offer a "finder's fee" of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually do my best to give you immortal glory, by publishing your name in the eventual book:–)

http://www-cs-faculty.stanford.edu/~knuth/fasc6a.ps.gz

* * *

So far there are 75 pages (37 for text, 13 for exercises, 25 for answers); 15 of those pages are about basic probability techniques, not about satisfiability.

I think the section on SAT will eventually amount to about 150 pages in Volume 4B. (For comparison, Volume 4A already contains about 135 pages about BDDs and ZDDs.)

bytes	date	program	
28336	2011-08-26	`sat0.w`	tabula rasa
29623	2011-09-03	`sat1.w`	truth table
22240	2011-09-13	`sat2.w`	incremental truth table
39733	2011-09-18	`sat3.w`	DPL 1962 “ready list”
39727	2011-11-24	`sat4.w`	prefer binary clauses
42912	2011-12-04	`sat5.w`	more efficient
55812	2011-12-28	`sat6.w`	Cook’s clause learning
28303	2012-02-06	`sat7.w`	Papadimitriou’s “Walk”
29380	2012-03-07	`sat8.w`	“WalkSAT”
38981	2012-05-10	`sat9.w`	Survey Propagation

It operates on an expanding set of clauses $C_1, C_2, \ldots, C_n$ (starting with the initial ones)

It has a push down stack of literals $\ell_1, \ldots, \ell_m$, consistent with no repetitions.

At every time we have a partial truth assignment obtained by assigning true to $\ell_1, \ldots, \ell_m$ on the stack. We think of the objective as finding a satisfying assignment.

1) $m \leftarrow 0$; $C_1, \ldots, C_n$ given (we start with initial clauses $C_1, \ldots, C_n$ and empty stack)

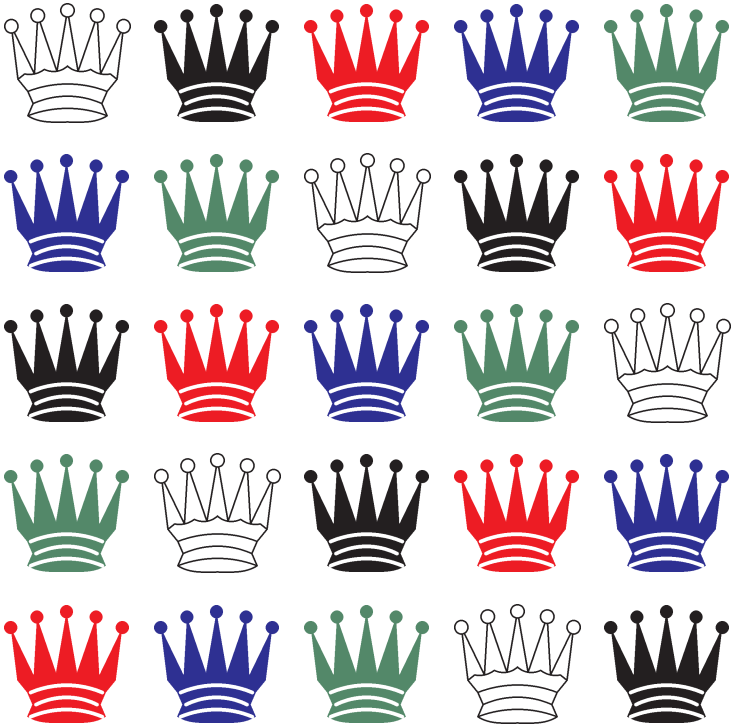
Comment:

At every time before we enter step 2) no clause is completely falsified by the partial truth assignment which verifies all $\ell_1, \ldots, \ell_m$.

2) $m \leftarrow m+1$

Select a literal ℓ_m such that neither ℓ_m nor $\overline{\ell}_m$ appears on the stack.

If no such ℓ_m exists then we have found a satisfying



```
0.0.1    0.0.2    0.0.3    0.0.4    0.0.5
0.1.1    0.1.2    0.1.3    0.1.4    0.1.5
0.2.1    0.2.2    0.2.3    0.2.4    0.2.5
         .    .     .    .    .
4.3.1    4.3.2    4.3.3    4.3.4    4.3.5
4.4.1    4.4.2    4.4.3    4.4.4    4.4.5
~0.0.1    ~1.1.1
~0.0.1    ~2.2.1
~0.0.1    ~3.3.1
         .    .     .    .    .
~4.2.5    ~4.3.5
~4.2.5    ~4.4.5
~4.3.5    ~4.4.5
```

http://www-cs-faculty.stanford.edu/~knuth/programs/sat-to-dimacs.w

http://www-cs-faculty.stanford.edu/~knuth/programs/dimacs-to-sat.w

```
c file created by SAT-TO-DIMACS Wed Jun 6 10:23:47 2012
c 4.4.5 -> 125
c 4.4.4 -> 124
   .  .  .  .
c 0.0.2 -> 2
c 0.0.1 -> 1
p cnf 125 825
-125 -120 0
-125 -115 0
   .  .  .  .
15 14 13 12 11 0
10 9 8 7 6 0
5 4 3 2 1 0
```

```
[1]> sat0 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~0.0.1 ~0.0.2 ~0.0.3 ~0.0.4 0.0.5 ... ~4.4.5
Altogether 19700+18785 mems, 39208 bytes.

[2]> sat1 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~0.0.1 0.0.2 ~0.0.3 ~0.0.4 ~0.0.5 ... 4.4.5
Altogether 12652+2709 mems, 40624 bytes.

[3]> sat2 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~0.0.1 ~0.0.2 ~0.0.3 ~0.0.4 0.0.5 ... ~4.4.5
Altogether 4526+3517938425 mems, 15516 bytes.
```

```
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
```

46. As an experiment, I'm swapping the first true literal into the first position of its clause, hoping that subsequent "decrease" loops will thereby be shortened.

```
⟨ Increase the breakcount of c's critical variable 46 ⟩ ≡
  {
    for (o, ii = i = cmem[c].start; ; i++) {
      o, q = mem[i];
      if (o, value(q)) break;
    }
    o, vmem[q ≫ 1].breakcount++;
    if i ≠ ii oo, mem[i] = mem[ii], mem[ii] = q;
  }
```

This code is used in section 44.

```
[4]> sat3 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~0.0.1 ~0.0.2 ~0.0.3 ~0.0.4 0.0.5 ... ~1.3.4
Altogether 19952+11647 mems, 42724 bytes.

[5]> sat5 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~0.0.1 ~0.1.1 ~0.2.1 ~0.3.1 0.4.1 ... 0.2.3
Altogether 18851+688489345 mems, 73248 bytes.

[6]> sat6 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
~1.1.1 ~2.2.1 ~3.3.1 ~4.4.1 ~0.1.1 ... 0.3.5
Altogether 0 new clauses, 0 recycled,
           36176+17047 mems, 74748 bytes.
```

```
[7]> sat3 d10000000 < rand-3-125-600-0.dat
(125 variables, 600 clauses, 1800 literals)
after 10002287 mems:00431033100502445415254544
after 20001484 mems:00432333523050144521515144
after 30002010 mems:03314040324303244155053 5
after 40000099 mems:30451003240345355541554544
after 50005241 mems:330204330400545431512 5
after 60000099 mems:33323533510505454044554 4
~
Altogether 27638+60489039 mems, 42124 bytes.
```

```
[8]> sat5 d10000000 < rand-3-125-600-0.dat
(125 variables, 600 clauses, 1800 literals)
after 10002965 mems:0acd545d54a456daa544c47ad
after 20003037 mems:3bcb44b554dc4babca
after 30000035 mems:3c5ba445a54bd4a4dd4544d44
~
Altogether 18551+31987450 mems, 69648 bytes.
```

```
[9]> sat3 d2000 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
after 2019 mems:11114555
after 4027 mems:111145555555555511145555
after 6026 mems:1111455555555555111455555555
after 8043 mems:1111455555555555111455555555
after 10032 mems:1111455555555555111455555555
~0.0.1 ~0.0.2 ~0.0.3 ~0.0.4 0.0.5 ... ~1.3.4
Altogether 19952+11647 mems, 42724 bytes.
```

```
[10]> sat5 d100000000 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
after 100000033 mems:bbbbbbbbbbbbbbc55555577
after 200005826 mems:bbbbbbbbbbbc555555557777
after 300000049 mems:bbbbbbbbbc5555557777bc55
after 400000033 mems:bbbbbbbc555555557777bbb
after 500000050 mems:bbbbbbc5555555557777bbb
after 600000005 mems:bbbbbc5555555557777bc55
~0.0.1 ~0.1.1 ~0.2.1 ~0.3.1 0.4.1 ... 0.2.3
Altogether 18851+688489345 mems, 73248 bytes.
```

```
[11]> sat3-rand d100000 < queen5x5color.dat
(125 variables, 825 clauses, 1725 literals)
after 100023 mems:11111111111111111111111114
after 202114 mems:11111111111111111111111125
after 300010 mems:11111111111111111111125555
after 400042 mems:11111111111111111112555555
after 501349 mems:11111111111111111125555555
after 603041 mems:11111111111111111255555555
after 701449 mems:11111111111111112555555555
~1.0.2 ~2.3.1 ~2.1.5 ~2.0.2 ... ~0.2.5
Altogether 21679+795425 mems, 42724 bytes.
```

```
10000 01000 00100 00010 00001
00010 00001 10000 01000 00100
10000 01000 00100 00010 00001
00001 10000 01000 00100 00010
00100 00010 00001 10000 01000
```

```
000   001   010   011   100
011   100   000   001   010
000   001   010   011   100
100   000   001   010   011
010   011   100   000   001
```

```
~0.0.3    ~0.0.1
~0.1.3    ~0.1.1
   .         .        .        .
~4.4.3    ~4.4.1
~0.0.2    ~0.0.1
~0.1.2    ~0.1.1
   .         .        .        .
~4.4.2    ~4.4.1
~0.0^0.11    0.0.1    0.1.1
0.0^0.11    0.0.1    ~0.1.1
0.0^0.11    ~0.0.1    0.1.1
~0.0^0.11    ~0.0.1    ~0.1.1
   .         .        .        .
0.0^4.41    0.0^4.42    0.0^4.43
```

```
[12]> sat5 d500000 < queen5x5color-binary.dat
(555 variables, 2130 clauses, 6340 literals)
after 500597 mems:ba45a45a4a55a55d554a5554a55
after 1000633 mems:ba45a45a4a55a55d554a5554a5
after 1500013 mems:ba45a45a4a55a55d554a5554a5
after 2000006 mems:ba45a45a4a55a55d554a5554a5
~0.0.3 0.0^4.43 4.4.3 ~4.4.1 ... ~1.3^1.41
Altogether 67011+2312306 mems, 256608 bytes.

[13]> sat3 d1000000000000 < queen5x5color-bin
(555 variables, 2130 clauses, 6340 literals)
after 1000000000063 mems:11151541551545424445
after 2000000000122 mems:11151541552445155515
~0.0.3 ~0.0.1 ~0.1.3 ~0.0^0.13 ... 0.2^0.32
Altogether 116372+2189929944193 mems, 156244
```

```
[14]> sat3-rand d1000000000000 < queen5x5colo
(555 variables, 2130 clauses, 6340 literals)
after 1000000001844 mems:00000000000000000000
after 2000000012419 mems:00000000000000000003
after 3000000002278 mems:00000000000000000003
after 4000000000002 mems:00000000000000000030
after 5000000000557 mems:00000000000000000030
after 6000000004086 mems:00000000000000000033
after 7000000011274 mems:00000000000000000033
after 8000000000156 mems:00000000000000000300
after 9000000000495 mems:00000000000000000303
after 10000000004416 mems:00000000000000000030
after 11000000000191 mems:00000000000000000033
etc... estimated 3 petamems to finish (3e15)!
```

```
~0.0.3  ~0.0.1
  .   .   .   .
~4.4.2  ~4.4.1
0.0.3  0.1.3  0.0.2  0.1.2  0.0.1  0.1.1
~0.0.3  ~0.1.3  0.0.2  0.1.2  0.0.1  0.1.1
0.0.3  0.1.3  ~0.0.2  ~0.1.2  0.0.1  0.1.1
~0.0.3  ~0.1.3  ~0.0.2  ~0.1.2  0.0.1  0.1.1
0.0.3  0.1.3  0.0.2  0.1.2  ~0.0.1  ~0.1.1
  .   .   .   .
0.0.3  4.4.3  0.0.2  4.4.2  0.0.1  4.4.1
~0.0.3  ~4.4.3  0.0.2  4.4.2  0.0.1  4.4.1
0.0.3  4.4.3  ~0.0.2  ~4.4.2  0.0.1  4.4.1
~0.0.3  ~4.4.3  ~0.0.2  ~4.4.2  0.0.1  4.4.1
0.0.3  4.4.3  0.0.2  4.4.2  ~0.0.1  ~4.4.1
```

```
[15]> sat5 < queen5x5color-altbinary.dat
(75 variables, 850 clauses, 4900 literals)
0.0.3 ~0.0.1 0.1.3 ~0.1.1 0.0.2 ... ~2.1.1
Altogether 43651+76175 mems, 145248 bytes.

[16]> sat3 < queen5x5color-altbinary.dat
(75 variables, 850 clauses, 4900 literals)
0.0.3 ~0.0.1 0.1.3 ~0.1.1 ~0.2.3 ... ~0.4.2
Altogether 77652+675724229162 mems, 90342 byt

[17]> sat3-rand < queen5x5color-altbinary.dat
(75 variables, 850 clauses, 4900 literals)
3.4.2 ~3.4.1 0.1.2 ~0.1.1 4.2.3 ... ~0.4.2
Altogether 78695+6426416 mems, 90324 bytes.
```

							x_5	x_4	x_3	x_2	x_1	x_0
					y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
1	1	0	0	1	0	0	0	0	0	0	0	1

$$(X \times Y = 6401)$$

```
~Z13
A2:6  ~Y4  ~X2
~P0:6  ~A1:6  ~A0:6
R0:6  ~A2:6  ~P0:6
Q1:9  ~A4:9  ~A3:9
~A9:11  R1:10  Q1:10
A10:5  A8:5  ~P2:5
P3:7  A10:7  ~A9:7
~R3:3  Q2:2
~U11  A13:11  P3:11
C8:3  ~C8:2
~C11:2  B11:2  V10
~D13:2  U12
  .   .   .   .
```

```
[18]> sat3 < 6401.dat
(286 variables, 911 clauses, 2309 literals)
~Z13 ... ~A8:9 A10:9 ~A6:8 ~Q2:8 ~A7:7 ~Y4
Altogether 40217+183094 mems, 63704 bytes.

[19]> sat5 < 6401.dat
(286 variables, 911 clauses, 2309 literals)
~Z13 ... A0:2 A2:2 ~P2:3 A0:5 X2 Y2
Altogether 26121+12307600 mems, 104784 bytes.

[20]> sat6 < 6401.dat
(286 variables, 911 clauses, 2309 literals)
~Z13 ... A2:5 P2:8 ~A10:8 ~A1:4 ~A1:6 ~B8:5
Altogether 2019 new clauses, 439 recycled,
           40415+3159955 mems, 720648 bytes.
```

```
[21]> sat5 d5000000 < 6401.dat
(286 variables, 911 clauses, 2309 literals)
after 5008203 mems:5445545555555444a4455d55\
     d55ba4b44d54d54d5455b55555555d545c5
after 10001285 mems:5445545555555444d54a445\
     5d555d55bd55555555d555ba4b44d54d54a445\
     5d555555c54a44555555555555455b555ba4d4\
     55b555555555555455555d5
~Z13 ... A0:2 A2:2 ~P2:3 A0:5 X2 Y2
Altogether 26121+12307600 mems, 104784 bytes.
```

```
[22]> sat3-rand d50000000 < 6401.dat
(286 variables, 911 clauses, 2309 literals)
after 50000824 mems:5445545555555444102455\
          5555555555455555555553535555555555\
          5554555555555555535513555155155544\
          4555555555554345555555555555511135\
          5555545355015355553555555555555555\
          510555511524
~Z13 ... ~A1:4 ~A1:3 ~A1:2 ~Q0:2 ~A1:1 ~R1:10
Altogether 44190+54916781 mems, 63704 bytes.
```

Cynthia A. Brown and Paul W. Purdom, Jr., *IEEE Transactions* **PAMI-4** (1982)

voted to making the program fast. For large problems, nearly all the time was spent doing intermediate predicate evaluations, so optimization of predicate evaluations received particular attention. Backtracking proceeds by setting or changing one variable at a time. For our purposes, a clause is true if at least one literal in it is true or unknown. We keep a list for each literal. Each clause is on the list for one of the literals that causes it to be true. Initially, when no variables are set, any literal in a clause causes it to be true. When a variable is set or changed, one literal becomes false. The clauses on the list for that literal are examined. If a clause contains other literals that cause it to be true, then the clause is moved to the list for that literal. If all the literals in a clause are false, then the predicate is false, and it is time to backtrack. After backtracking, the clause that caused the predicate to become false is again true, so it is not necessary to move it (or any other clauses remaining on its list) to a new list. Also, there is no need to move any other clauses when backtracking. Since a clause can be left on the list of *any* literal that makes it true, there is no need to restore other clauses to their former lists. This method of evaluating intermediate predicates allows us to evaluate large predicates rapidly. For $v = 256$, $t = 4096$, and $s = 3$, we estimate that only 35 μs are required for a typical evaluation.

I haven't written any of this stuff up yet.

But the current draft has lots of cool material about Monte Carlo / Las Vegas approaches to SAT solving, like WalkSAT.

Here's an excerpt that you might find interesting:

Exercise 175 proves that such an algorithm always has an optimum cutoff value $N = N^*$, which minimizes the expected time to success when the algorithm is restarted after each failure. Sometimes $N^* = \infty$ is the best choice, meaning that we should always keep plowing ahead; in other cases N^* is quite small.

But N^* exists only in theory, and the theory requires perfect knowledge of the algorithm's behavior. In practice we usually have little or no information about how N should best be specified. Fortunately there's still an effective way to pro-

ceed, by using the notion of *reluctant doubling* introduced by M. Luby, A. Sinclair, and D. Zuckerman [*Information Proc. Letters* **47** (1993), 173–180], who defined the interesting sequence

$$S_1, S_2, \ldots = 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,\\ 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,16,\ldots \quad (97)$$

The elements of this sequence are all powers of 2; and we have $S_{n+1} = 2S_n$ if and only if the number S_n has already occurred an even number of times. A convenient way to generate this sequence is to work with two integers (u, v), and to start with $(u_1, v_1) = (1, 1)$; then

$$(u_{n+1}, v_{n+1}) = \big(u_n \,\&\, {-u_n} = v_n ?\, (u_n + 1, 1) : (u_n, 2v_n)\big). \quad (98)$$

The successive pairs are $(1,1)$, $(2,1)$, $(2,2)$, $(3,1)$, $(4,1)$, $(4,2)$, $(4,4)$, $(5,1)$, $\ldots$, and we have $S_n = v_n$ for all $n \geq 1$.

Exercise 178 introduces the "reluctant Fibonacci sequence"

$$1, 1, 2, 1, 2, 3, 1, 1, 2, 3, 5, 1, 1, 2, 1, 2, 3, 5, 8,$$
$$1, 1, 2, 1, 2, 3, 1, 1, 2, 3, 5, 8, 13, 1, \ldots$$

and its answer explains how to generate it with an amazingly short sequence of bitwise operations.

185. [*46*] If the given clauses are satisfiable, and if $p > 0$, can there be an initial x for which Algorithm W always loops forever?

(answer)

185. *Note to the reader:* I've been tearing my hair out trying to resolve this question. If there is a counterexample, I definitely ought to mention that fact in my book. If not, the result still is essential for a basic understanding of this important algorithm. I've been alternating between thinking I had a counterexample and thinking that there is a simple way to prove that no counterexamples exist. I can't figure out the essential reason why counterexamples have escaped all of my devious constructions. Hopefully somebody will already have resolved this issue? Help!

185. [*46*] If the given clauses are satisfiable, and if $p > 0$, can there be an initial x for which Algorithm W always loops forever?

(revised answer)

185. Equivalently, consider the following digraph on 2^n vertices, one vertex for each $x = x_1 \ldots x_n$: For all clauses C_j not satisfied by x, there's an arc from x to all vertices y that are obtainable by a flip that Algorithm W might make. (In particular, if at least one literal of C_j has cost zero, the literals of nonzero cost are *not* flipped.) Does every strong component of this digraph have an exit, unless it represents a solution?

185. [*36*] (H. H. Hoos, 1998.) If the given clauses are satisfiable, and if $p > 0$, can there be an initial x for which Algorithm W always loops forever?

(current answer)

185. (Solution by Bram Cohen, 2012.) Consider the 10 clauses $\bar{1}\bar{2}34\bar{5}67$, $\bar{1}2\bar{3}4\bar{5}67$, $123\bar{4}5$, $123\bar{4}6$, $123\bar{4}7$, $\bar{1}\bar{2}\bar{3}\bar{4}$, $\bar{1}\bar{2}\bar{3}\bar{5}$, $\bar{1}\bar{2}\bar{3}\bar{6}$, $\bar{1}\bar{2}\bar{4}\bar{5}$, $\bar{1}\bar{2}\bar{4}\bar{6}$, and 60 more that are obtained by the cyclic permutation (1234567). All binary $x = x_1 \dots x_7$ with weight $\nu x = 2$ have cost-free flips leading to weight 3, but no such flips to weight 1. Since the only solution has weight 0, Algorithm W loops forever whenever $\nu x > 1$. (Is there a smaller example?)

AND NOW
A SPECIAL REQUEST

WHAT IS YOUR FULL NAME?

http://www-cs-faculty.stanford.edu/
~knuth/
fasc6a.ps.gz

Thanks for listening!