EPiC Series in Computer Science

Volume 36, 2015, Pages 14–26

GCAI 2015. Global Conference on Artificial Intelligence

(The Final) Countdown

Jean-Marc Alliot and Charlie Vanaret

Institut de Recherche en Informatique de Toulouse

Abstract

The Countdown game is one of the oldest TV show in the world. It started broadcasting in 1972 on the French television and in 1982 on British channel 4, and it has been running since in both countries. The game, while extremely popular, never received any serious scientific attention, probably because it seems too simple at first sight. We present in this article an in-depth analysis of the numbers round of the Countdown game. This includes a complexity analysis of the game, an analysis of existing algorithms and the presentation of a new algorithm that increases resolution speed by a large factor. It also includes some leads on how to turn the game into a more difficult one, both for a human player and for a computer, and even to transform it into a possibly undecidable problem.

1 Introduction

The numbers round of the Countdown game is extremely simple: 6 numbers are drawn from a set of 24 which contains all numbers from 1 to 10 (small numbers) twice plus 25, 50, 75 and 100 (large numbers). Then, with these six numbers, the contestants have to find a number randomly drawn between 101 and 999 or, if it is impossible, the closest number to the number drawn. Only the four standard operations ($+ \; - \; \times \; /$) can be used. As soon as two numbers have been used to make a new one, they can't be used again, but the new number found can be used. For example, if the six numbers drawn are *1,1,4,5,6,7* and the number to find is *899* the answer is:

```
  Operations   Remaining
 6 x  5 = 30  {1,1,4,7,30}
30 +  1 = 31  {1,4,7,31}
 4 x  7 = 28  {1,28,31}
28 +  1 = 29  {29,31}
29 * 31 = 899 {899}
```

There was a very early article in the French magazine "l'Ordinateur Individuel" in the late seventies, written by Jean-Christophe Buisson [1], which described a simple algorithm to solve the game. The only article written on the subject in English was published twice [2, 4] by Daniel Defays. Defays also published in 1995 a book in French [3] which used the game as a central example for introducing artificial intelligence methods. But the ultimate goal of Defays was not to develop an accurate solver for the game, but a solver mimicking human reasoning (such as the *Jumbo* program by Hofstadter [6]), including possible mistakes. Many websites [12] in France and in Great Britain discuss the game and how to program it, with lots of statistics

and sometimes lots of errors. This paper presents an in-depth analysis of the game and of its extensions.

In this paper: $n!$ is the factorial of n, $\binom{n}{p} = \frac{n!}{p!\,(n-p)!}$ is the number of subsets having p elements in a set of n distinct elements, $E(x)$ is the integer part of x.

2 Elementary algorithms

2.1 Decomposition in sub-problems

The first published algorithm [1] used a simple decomposition mechanism. Let's consider the following example: numbers *3, 50, 7, 4, 75, 8*, number to find *822*. The algorithm would start from the solution (822) and use a backward chaining approach in the Prolog way. However, not all operations were tried; at odd steps, only addition and subtractions were used, while at even steps only divisions were used. So the algorithm would generate thirteen numbers at the first step: 822, 822 ± 3, 822 ± 50, ..., and then try to divide all of them by the remaining 5 (or 6 if no number was added or subtracted) numbers. If a division succeeds, the algorithm would then be applied recursively on the new result with the remaining numbers. When 0 is reached the solution has been found.

This algorithm has serious drawbacks. It is impossible to compute solutions requiring intermediate results, such as the first one presented in this article, because 31 and 29 must be built independently before multiplying them to have 899. It is even impossible to find solutions with divisions. This algorithm was later refined with faster machines by using all possible operations at each step. The maximal complexity of the algorithm is now $(6\times4)\times(5\times4)\times\cdots(1\times4) = 6!\,4^6$ If we consider the general case with n numbers the complexity is $n!\,4^n$. For $n = 6$, the maximal number of operations is 491520. If we consider that the actual number of operations at each step is closer to 3 than to 4, we have a minimal complexity of $n!\,3^n$, and for $n = 6$ the minimal number of operations is 87480. It is still impossible to find solutions requiring intermediate results, and impossible to find directly approximate results.

2.2 The depth first algorithm

Let's consider the complete set of n numbers. We pick two of them ($\binom{n}{2} = \frac{n(n-1)}{2}$ possibilities) and combine them using one of the four possible operations. The order of the two numbers picked is irrelevant as the order does not matter for the addition and the multiplication ($a+b = b+a$ and $a\times b = b\times a$), and we can only use one order for the other two operations (if $a > b$, we can only compute $a-b$ and a/b, and if $a < b$, $b-a$ and b/a). The result is put in the set, giving a new set of $n-1$ numbers. This operation is repeated until no number remain in the pool and it then backtracks to the previous point of choice (number or operation). This is a simple depth-first search algorithm, which is exhaustive as it searches the whole computation tree.

The maximal complexity of the algorithm is given by: $(\frac{n\times(n-1)}{2}\times4)\times(\frac{(n-1)\times(n-2)}{2}\times4)\times\cdots\times(\frac{2\times1}{2}\times4)$. This gives:

$$d_{max}(n) = n!\,(n-1)!\,2^{n-1} \quad (1)$$

$$d_{min}(n) = n!\,(n-1)!\,(\frac{3}{2})^{n-1} \quad (2)$$

For $n = 6$, we have a maximal number of 2764800 operations and a minimal number of 656100 operations. The algorithm is easy to implement, no complex data structures are needed, and

it requires almost no memory.

2.3 The breadth first algorithm

The breadth first algorithm is a little bit more difficult to understand. It is also a recursive algorithm, but it works on the partitions of the set of numbers. The first presentation of this algorithm seems to be [11]. Here $g(\{a_1,\cdots,a_n\})$ is the set of numbers generated by numbers $\{a_1,\cdots,a_n\}$ using the four operations.

- First, we create all sets generated by only one element. With the same example, we have of course 6 elements $g(\{3\})=\{3\}$, $g(\{50\})=\{50\}$, $g(\{7\})=\{7\}$, $g(\{4\})=\{4\}$, $g(\{75\})=\{75\}$, $g(\{8\})=\{8\}$
- Next we create the sets of all numbers that can be computed using only two numbers. Here for example all the numbers generated by $\{3,50\}$ are the elements of $g(\{3\})$ applied to the elements of $g(\{50\})$ which give the set $g(\{3,50\})=g(\{3\}).g(\{50\})=\{53,47,150\}$. $\{3\}$ and $\{7\}$ give $g(\{3,7\})=\{10,4,21\}$. $\{50\}$ and $\{7\}$ give $g(\{50,7\})=\{57,43,350\}$. We will have $\binom{6}{2}$ such sets.
- Next we create the sets of all numbers that can be computed using only 3 numbers. For example, the set of numbers generated by the 3 numbers 3, 50 and 7 is $g(\{3,50,7\})=g(\{3\}).g(\{50,7\})\cup g(\{50\}).g(\{3,7\})\cup g(\{7\}).g(\{50,3\})$ Here for example $g(\{3\}).g(\{50,7\})=\{54,60,171,19,40,46,129,347,353,1050\}$ There are $\binom{6}{3}$ such sets.
- The algorithm proceeds with all sets generated by 4 numbers. There are $\binom{6}{4}$ such sets.
- There are $\binom{6}{5}$ sets generated by 5 numbers.
- Then the set generated by all six numbers is created.

The complexity of this algorithm is not so easy to compute. It is sometimes mistakenly presented as being 2^n [9], but it is a very crude estimation.

If we call $N(p)$ the number of elements in a set generated by p elements, the total number of operations will be $\sum_{p=1}^{n}\binom{n}{p}N(p)$. It is possible to establish a recurrence relationship between $N(p)$ and $N(p-1)$, $N(p-2)$, etc. For example, $N(4)$ is the sum of two terms:

- $N(3)\times N(1)\times 4\times\binom{4}{3}$ which is the number of elements in a set built by combining with the 4 operations a set having $N(1)$ elements and a set having $N(3)$ elements. There are $\binom{4}{3}=4$ such numbers. For example for $\{1,2,3,4\}$, we have $\{1,2,3\}.\{4\}$, $\{1,2,4\}.\{3\}$, $\{1,3,4\}.\{2\}$, $\{2,3,4\}.\{1\}$
- $N(2)\times N(2)\times 4\times(\binom{4}{2}/2)$ For example, for $\{1,2,3,4\}$ we combine $\{1,2\}.\{3,4\}$, $\{1,3\}.\{2,4\}$ and $\{1,4\}.\{2,3\}$

More generally, we have:
$N(p)=(\sum_{i=1}^{p-1}\binom{p}{i}N(i)N(p-i))/2\times 4$
A simple computation gives:
$N(p)=4^{p-1}\prod_{i=1}^{p-1}(2i-1)$
And thus the complexity for n numbers is:

$$b_{max}(n) = \sum_{p=1}^{n}\binom{n}{p}4^{p-1}\prod_{i=1}^{p-1}(2i-1) \quad (3)$$

$$b_{min}(n) = \sum_{p=1}^{n}\binom{n}{p}3^{p-1}\prod_{i=1}^{p-1}(2i-1) \quad (4)$$

For $n = 6$ we have a maximal number of 1144386 operations, half the number of the operations required by the depth first algorithm, and a minimal number of 287331 operations.

3 Implementation and refinements

To compare the algorithms, the programs were all written in Ocaml [7]. The implementation was not parallel and the programs were run on a Intel 980X processor. For very large instances, an implementation of the best algorithm (depth first with hash tables) was written in C (with the same algorithm, the C program on a single core is twice faster than the Ocaml program). MPI [10] was used to solve problems in parallel and the program was run on the OSIRIM cluster at IRIT (a 640 AMD-HE6262 cores cluster) using 512 cores.

In this section we study the standard Countdown game: $n = 6$ numbers are drawn from a pool of 24, with all numbers in the range 1-10 present twice, plus one 25, one 50, one 75 and one 100. The number of different possible instances is:

$$\binom{14}{6}(\text{no pair})+\binom{10}{1}\times\binom{13}{4}(\text{one pair})+\binom{10}{2}\times\binom{12}{2}(\text{two pairs})+\binom{10}{3}(\text{three pairs}) = 13243$$

Programs are so fast that trying to accurately measure the execution time of a single instance is impossible. So, in the rest of this section, all programs solve the complete set of instances and the time recorded is the time to complete the entire set: when a time of 160s is given, the mean time of resolution of one instance is $160/13243 = 0.01s$

3.1 The depth first algorithm with hash tables

This algorithm is the same as the depth first algorithm described in section 2.2. The idea is to use an (old) [13] improvement which has been often used in many classical board games: hash tables.

When solving the game, if the same set of numbers appears a second time in the resolution tree, the branch can be discarded: as it is a depth first search where the size of the set of numbers strictly decreases by one at each level in the tree, we know that this branch has already been fully developed somewhere else in the tree and that all possible results have already been computed and all numbers that can be found with that set of numbers have already been marked. We just need a way to uniquely identify an identical set of numbers Thus an array $h(x)$ of 60 bits random values is created at the start of the program. Each time a number x is added to the pool of numbers, $h(x)$ is added to the hash value, and when x is removed from the pool, $h(x)$ is subtracted from the hash value. Values are stored in an array, and a mask of n-bits is applied to the 64-bits hash value, returning an index for this array (the size of the array is of course 2^n). Hash collisions happen when two different objects having different hash values have the same hash index. They can be solved in two ways: maintaining a set of values for each array element, or having a larger array to minimize hash collisions. Having a too large array can also have detrimental consequences as the access to the hash array is mostly random, cache faults are very likely to happen at each access if the array doesn't fit in the cache. On figure 1, we have the result of the experimentation. The x-axis is the size of the hash table in bits, the y-axis the time needed to solve the 13243 instances. The blue plot is the time without hash tables, the red one the time with a simple array hash table and the green one the time with an array containing sets to hold all numbers.

The green plot shows that when we store all results in an array of sets, there are quickly too many elements, and thus we are never able to remain inside the L2 cache. For $n = 15$, the

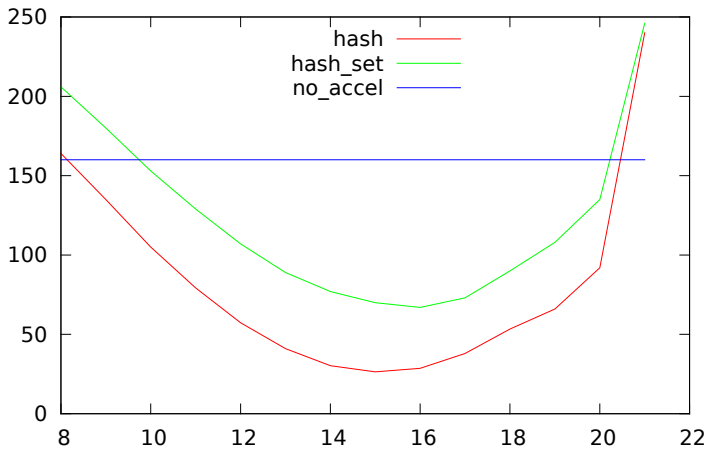

Figure 1: Time used vs size. x-axis: size of the hash table (from 2^8 to 2^{22}), y-axis:time

13243 instances are solved by the array-hash method in 26s, 5 times faster than without hash tables.

3.2 The breadth first algorithm

A data structure that contains the information needed to build the numbers generated by a subset of the initial pool is used. An array of list is used where the i-th element contains the list of pairs of sets to combine in order to build the numbers generated by the subset represented by the binary decomposition of i. For example:

- For $i = 16$ we have $i = 16 = 10000_2$, so this element will just point to the fifth element in the initial pool of numbers.
- The element at $i = 5 = 101_2$ points to the list of pairs of sets to combine. Here, we have to combine with the four operations the first element and the third element of the original pool, so there is only one pair $(1, 3)$.
- The element at $i = 25 = 11001_2$ will contain the pairs $(1, 24)$, $(8, 17)$ and $(9, 16)$ because to have all elements generated by the first, the fourth and the fifth element of the original pool we have to combine with the four operations (a) all elements generated by the fourth and the fifth with the first element, (b) all elements generated by the first and the fourth with the fifth element and (c) all elements generated by the first and the firth with the fourth element.

This array of list of pairs can be pre-computed and stored once and for all. The size of the array is $2^n - 1$ where $n = 6$, so the array here has 63 lists of pairs. Another array of the same size is used, where the i-th element is an array that will hold all numbers generated for the i index. For example, if the initial pool of numbers is $\{7, 8, 9, 10, 25, 75\}$ we first copy 7 at position 1, 8 at position 2, 9 at position 4, 10 at position 8, 25 at position 16 and 75 at position 32: all elements with an index having only 1 bit are filled. Then we fill all elements having an index with 2 bits. For example, element $3 = 11_2$ is $\{7 + 8, 8 - 7, 7 \times 8\} = \{15, 1, 56\}$, element

$5 = 101_2$ is $\{7+9, 9-7, 9\times 7\} = \{16, 2, 63\}$, element $6 = 110_2$ is $\{17, 1, 72\}$, and so on. When all elements with a 2-bits index are filled, elements with a 3-bits index are filled.

To store the results, we can use either an array of arrays or and array of sets. Experimental results with $n = 6$ are summarized in table 1. To fully appreciate the efficiency of the program,

Algorithm	Total Time	By instance
Depth first	160	12.10E-3
Depth first / hash	26	1.96E-3
Depth first / hash-set	67	5.05E-3
Breadth first / arrays	53	4.00E-3
Breadth first / sets	89	6.72E-3

Table 1: Comparison of the algorithms implemented in Ocaml ($n = 6$, 13243 instances solved)

there are other programs available on the net which claim to solve also all the instances such as [5], but in 60 days, to be compared to 26s. There is another site [8] in French which advertises the kitsune program and gives also some stats. However, it takes a few hours to compute them.

4 Making the problem more difficult

There are different ways to change the difficulty of the game. If we choose to work with only 6 numbers it is possible to choose only tuples (numbers set,target value), such as the number of operations for finding the target with the given numbers set is high. It is also possible to pick the 6 numbers in larger sets, for example all numbers between 1 and 100, with numbers less "simple" to manipulate (such as large primes). The other way is to increase the size of the set of numbers and to choose a target in a higher range.

To compute the total number of different instances when picking n numbers[1], we can extend the formula in section 3:

$$\binom{14}{n} + \ldots + \binom{10}{i} \times \binom{14-i}{n-2i} + \ldots + \binom{10}{E(n/2)}\binom{14-E(n/2)}{n-2E(n/2)} = \sum_{i=0}^{E(n/2)} \binom{10}{i} \times \binom{14-i}{n-2i}$$

This formula is valid for $n \leq 20$ and the number of instances is $n(7) = 27522$, $n(8) = 49248$, $n(9) = 76702$, and $n(10) = 104753$.

4.1 Solving for $n = 6$

4.1.1 Standard game

For $n = 6$ we have 13243 possible sets. In the standard numbers round of the Countdown game, we search for numbers in the range 101–999, so there are $899 \times 13243 = 11905457$ possible problems. 10858746 games are solvable (91.2%), 743896 problems (6.25%) have a solution at a distance of 1 (the nearest number). 1226 instances out of 13243 (9.2%) solve all target numbers in the range 101-999. Figure 2 represents the percentage of instances finding a specific target. The timings are available in table 1.

[1] The number of possible instances is not an indicator of the difficulty of the game, but we need these numbers in the next sections.

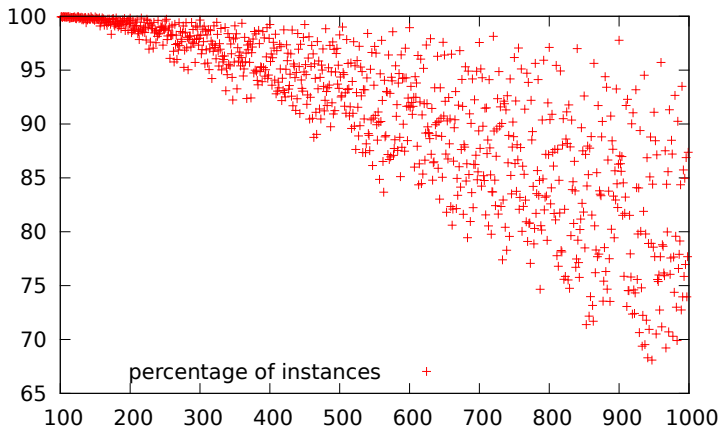

Figure 2: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 6$)

4.1.2 Selecting problems

The program creates a complete database that can be stored. For a given number set and a given target number we know if it can be solved and how many operations are necessary to solve it, or how close is the nearest findable number when it can't be solved. It is thus easy to select only interesting problems. There can be many different selection criteria: solvable problems requiring more than 4 (or 5...) operations, or unsolvable problems with the nearest number at a minimal given distance, or unsolvable problems with the nearest number requiring more than 4 operations, etc... This would turn the number round in something worth watching again.

4.1.3 Using a larger set to pick numbers

Another way to make the game harder would be to use all available numbers between 1 and 100 when picking the set. Building the full database is much more computing intensive. In the standard game we have 13243 sets, when picking k numbers between 1 and n (including repetitions) we have $\binom{n+k-1}{k} = \binom{100+6-1}{6} = 1609344100 \simeq 1.6\,10^9$ possible sets, and $1446800345900 \simeq 1.4\,10^{12}$ problems. Building the database took 12 hours on the cluster described in section 3.

In figure 3, we have the same results as in figure 2. Percentages are higher which means that on the average, the problem is easier to solve with numbers picked randomly between 1 and 100.

There are 73096123 (4.5%) sets that solve all problems. This is less ($1226/13243 \simeq 9.2\%$) in percentage than for the standard game, but there are 60000 times more sets if we consider the raw numbers. So we can select some sets with specified characteristic that would make them difficult for human beings, while maintaining the diversity of the problem. There are for example 52253 sets that solve all problems while being composed only by prime numbers, or 22136 sets by primes ≥ 3.

Another criteria could be to select sets where all numbers are greater than a given one; there are for example 20602 sets with all numbers > 25 that solve the 899 problems. This method

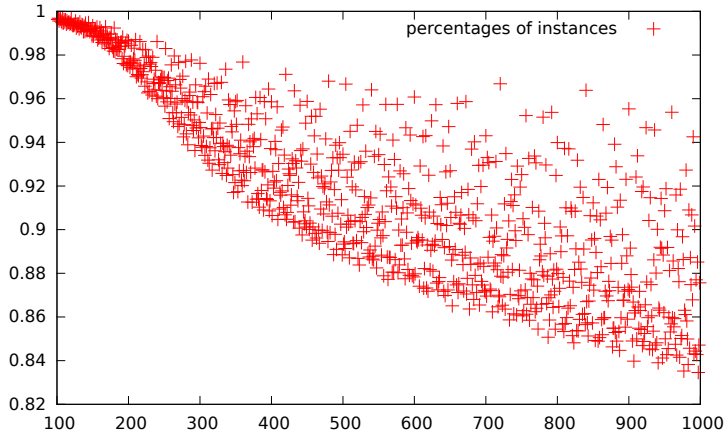

Figure 3: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 6$, extended set)

can be combined with the one described in section 4.1.2, by choosing only target numbers that require a minimum number of operations. Here again, the possibilities are endless, and it would turn the numbers game into something really difficult while always using 6 numbers.

4.2 Solving for $n = 7$

Using equations 1, 2, 3 and 4 we find that $\frac{d_{max}(7)}{d_{max}(6)} = 84$, $\frac{d_{min}(7)}{d_{min}(6)} = 63$, $\frac{b_{max}(7)}{b_{max}(6)} = 43$, $\frac{b_{min}(7)}{b_{min}(6)} = 32$.

In table 2 we have the results of the experimentation with the five algorithms with $n = 7$.

Algorithm	Time by instance
Depth first	740E-3
Depth first / hash	36E-3
Depth first / hash-set	114E-3
Breadth first / arrays	109E-3
Breadth first / sets	131E-3

Table 2: Comparison of the algorithms for $n = 7$

Regarding the resolution of problems we see on figure 4 how numbers are found. With an extra number in the set, the success rate becomes extremely high. All numbers are found by at least 98.5% of the instances: the problem has become too easy. The right solution is to look for target numbers in the range 1000–6000. The success rate is now almost the same as what it was with 6 numbers in the range 100–1000, but with a resolution time which is 20 times higher.

4.3 Solving for $n = 8$

We have here $\frac{d_{min}(8)}{d_{min}(7)} = 84$ and $\frac{b_{min}(8)}{b_{min}(7)} = 39$.

In table 3 we have the results of the experimentation for the five algorithms with $n = 8$.

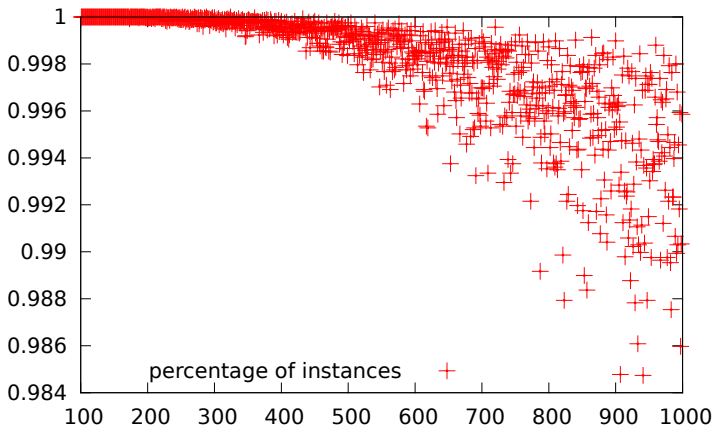

Figure 4: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 7$)

Algorithm	Time by instance
Depth first	61
Depth first / hash	1.2
Depth first / hash-set	3.7
Breadth first / arrays	4.4
Breadth first / sets	4.1

Table 3: Comparison of the algorithms for $n = 8$

The results are presented in figure 5. Computation took a few hours. There again, with an

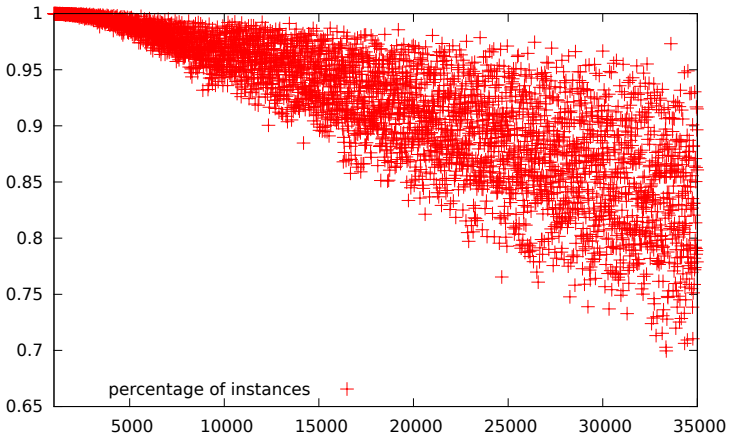

Figure 5: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 8$)

additional number, the problem becomes too easy to solve in the previous range (1000–10000). The range of the target number must be extended up to 35000 as we have then roughly the same average success rate as with the standard game.

4.4 Solving for $n = 9$

We have $\frac{d_{min}(9)}{d_{min}(8)} = 108$ and $\frac{b_{min}(9)}{b_{min}(8)} = 48$. Thus the standard depth first algorithm requires more than 6000s to solve a single instance and the breadth first algorithm with arrays needs around 40Gb of memory, that the computer used for these tests didn't have. In table 4 we have the results of the experimentation with three algorithms with $n = 9$.

Algorithm	Time by instance
Depth first	-
Depth first / hash	14.7
Depth first / hash-set	54.3
Breadth first / arrays	-
Breadth first / sets	46.7

Table 4: Comparison of the algorithms for $n = 9$

The results are presented in figure 6. We have to extend the range of the target number

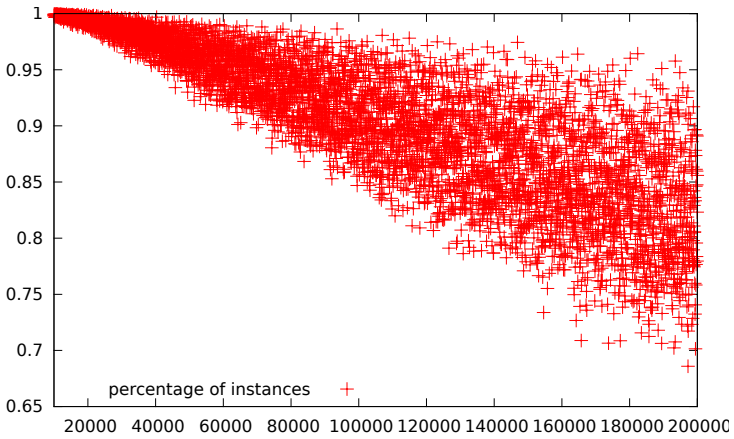

Figure 6: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 9$)

up to around 200000 (the most difficult number to find is 190667 with a success rate of 66%). Computing complete results took 3 days.

4.5 Solving for $n = 10$

For $n = 10$ we are at last entering uncharted territory. Solving one instance of the problem takes from a few seconds to a few minutes, so it seems impossible to use an exhaustive algorithm. We are at last back in the heuristics land. Complete results were computed in 20 hours on the

512 cores cluster described in section 3. We had to extend the range of the target number over 1000000 to have similar results regarding success rate. The results are presented in figure 7.

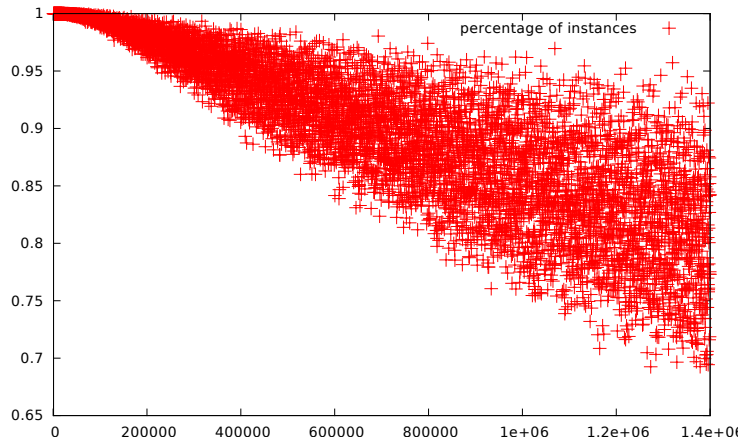

Figure 7: x-axis: number to find, y-axis:percentage of instances finding this number ($n = 10$)

5 A slightly modified problem

An other solution to turn the game into a more interesting one would be to add a simple operation: the possibility to replace any available number by its square. We now have long computations: to find 862 using the {1,10,10,25,75,100}, the shortest computation requires 14 steps:

```
10 - 1 = 9                        {9,10,25,75,100}
100 x 100 = 10000                 {9,10,25,75,10000}
9 x 9 = 81                        {81,10,25,75,10000}
10 x 10 = 100                     {81,100,25,75,10000}
100 x 100 = 10000                 {81,10000,25,75,10000}
10000 + 10000 = 20000             {81,20000,25,75}
75 x 75 = 5625                    {81,20000,25,5625}
5625 x 5625 = 31640625            {81,20000,25,31640625}
20000 x 20000 = 400000000         {81,400000000,25,31640625}
400000000 - 31640625 = 368359375 {81,368359375,25}
25 x 25 = 625                     {81,368359375,625}
625 x 625 = 390625                {81,368359375,390625}
368359375 / 390625 = 943          {81,943}
943 - 81 = 862                    {862}
```

The decidability of the game is now unclear. We have to set an upper bound A above which we do not square numbers anymore; without this bound, the algorithm would not stop for unsolvable instances. The 13 sets and 49 associated instances not solved for $A = 45000$ are:

```
1 1 10 10 25 100: 858
1 1 10 10 25  75: 863
1 1 10 10 50 100: 433 453 547 683 773 853
```

A	Sets	Instances	% unsolved
1	12017	1046711	8.79%
5	5004	128631	1.08%
10	593	7231	0.0607%
100	20	77	0.000647%
2000	16	54	0.000454%
10000	14	51	0.000428%
50000	13	49	0.000412%

Table 5: Number of sets having at least one instance not solved and unsolved instances as a function of A

```
1 1 10 10 50  75: 793 853 978
1 1 10 10 75 100: 433 453 457 478 547 618 653 682 708 718
                  778 793 822 853 892 907 958 978
1 1 10 25 75 100: 853 863
1 1 10 50 75 100: 793 813 853 978
1 1  5  5 25 100: 813 953
1 1  7  7 50 100: 830
1 1  8  8  9   9: 662
1 1  9 10 10 100: 478 573 587 598
1 1  9  9 10 100: 867
1 9  9 10 10 100: 867 947 957 958 967
```

Almost all problems are solvable when using the square operation with target numbers in the 101–999 interval (at least 99.999588% of them). The status of the 49 unsolved instances remains open. For targets in the 1001-9999 interval, the success rate is at least 99.9705%.

6 Conclusion

To turn the problem into a challenging one for a human being, this article proposes different solutions. As the game has been completely solved for $n = 6$, both with the standard set of numbers and with the extended set of all numbers from 1 to 100, it is easy to pick numbers and targets such that the problem is difficult for a human being, either by choosing problems which require a minimal number of operations, or unsolvable problems with the best findable number at some distance of the target, or sets having only prime numbers or large and "unfriendly" numbers. Another solution would be to use more than 6 numbers, and to use a target in a range above 1000, but it is probably not necessary. The last solution is to change a little bit the game by adding the *square* operation, and setting a target in the interval 1001-9999. While the theoretical success rate is very high, the problem is much more difficult for a human being, because the target is higher, and the square is not a natural operation to use.

It is more difficult to turn the game into a challenging problem for a computer. While the classical depth-first algorithm fails to find a solution in the allotted amount of time for $n > 7$, our algorithm solves the problem with up to 9 numbers in the set. The $n = 10$ problem is out of reach for an ordinary computer. It would however be interesting to start a challenge between computers for $n = 10$, or $n = 11$ to see what heuristics methods are the best for solving this problem.

Using the square operation fundamentally changes the problem from a theoretical point

of view, because the game is probably undecidable. Proving the undecidability remains an open challenge, and finding solutions for the currently unsolved problems (49 instances for the standard set of numbers and the standard target number range) is also still open.

References

[1] Jean-Christophe Buisson. A moi compte, deux mots! *L'ordinateur individuel*, 20, 1980.

[2] Daniel Defays. Numbo: A study in cognition and recognition. *The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 7(2):217–243, 1990.

[3] Daniel Defays. *L'esprit en friche: les foisonnements de l'Intelligence Artificielle*. Pierre Mardaga, 1995. ISBN: 2-87009-326-8.

[4] Daniel Defays. Numbo: A study in cognition and recognition. In Douglas Hofstadter, editor, *Fluid concepts and creative analogies: computer models of the fundamental mechanisms of thought*. BasicBooks, 1995.

[5] Patrice Fouquet. Le compte est bon. http://patquoi.free.fr/lcpdb/, March 2010.

[6] Douglas Hofstadter. *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*, chapter The Architecture of Jumbo. Harvester Wheatsheaf, 1995.

[7] INRIA. Ocaml. http://caml.inria.fr/ocaml/index.en.html, 2004.

[8] Julien Lemoine and Simon Viennot. Kitsune. http://kitsune.tuxfamily.org/wiki/doku.php, 2012.

[9] Jacky Mochel. Le compte est bon. http://j.mochel.free.fr/comptebon.php, April 2003.

[10] MPI-board. Mpi-2. http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm, 1997.

[11] Jean-Eric Pin. Le compte est bon. Sujets de projets 97-98 de tronc commun informatique de l'Ecole Polytechnique de Paris, June 1998. Institution: Laboratoire d'Informatique Algorithmique: Fondements et Applications.

[12] Williams Tunstall-Pedoe. Number games solver faq. "http://www.crosswordtools.com/numbers-game/faq.php#stats", 2013.

[13] Albert L. Zobrist. A new hashing method with application for game playing. Technical report 88, University of Wisconsin, Computer Science Department, April 1970.