**Noname manuscript No.**
(will be inserted by the editor)

# Snakes, coils, and single-track circuit codes with spread $k$

SIMON HOOD · DANIEL RECOSKIE · JOE SAWADA · DENNIS WONG

Received: date / Accepted: date

**Abstract** The snake-in-the-box problem is concerned with finding a longest induced path in a hypercube $Q_n$. Similarly, the coil-in-the-box problem is concerned with finding a longest induced cycle in $Q_n$. We consider a generalization of these problems that considers paths and cycles where each pair of vertices at distance at least $k$ in the path or cycle are also at distance at least $k$ in $Q_n$. We call these paths $k$-snakes and the cycles $k$-coils. The $k$-coils have also been called circuit codes. By optimizing an exhaustive search algorithm, we find 13 new longest $k$-coils, 21 new longest $k$-snakes and verify that some of them are optimal. By optimizing an algorithm by Paterson and Tuliani to find single-track circuit codes, we additionally find another 8 new longest $k$-coils. Using these $k$-coils with some basic backtracking, we find 18 new longest $k$-snakes.

**Keywords** snake, coil, circuit code, single-track, snake-in-the-box, longest path

## 1 Introduction

The *snake-in-the-box* and *coil-in-the-box* problems present difficult challenges to mathematicians and computer scientists regarding the longest induced paths or cycles in a hypercube that can be found. These problems have been heavily studied over the past 50 years [23,9,36,14,27,44,33,28,3,41,34,40,5,42,34,6,22,12,20,31,8,10,13,17, 21,26,25,30,35,39,49,7,48,4,46,24,45,37] including many dedicated to improving bounds [11,1,43,2,38,15, 47,16,29]. They were first described by Kautz [23] in relation to a theory of error correcting codes, but since have appeared in many other applications including electrical engineering, coding theory, combination locking, analog-to-digital conversion and network topology. Generally, the longer the snake or coil, the more useful it is [27] while the greater the spread, the greater the error detection capability [19]. Finding the longest induced paths, however, is well known to be NP-complete even when graphs are bipartite [18].

The $n$-dimensional hypercube, denoted $Q_n$, is the graph whose vertices consist of all binary strings $b_n \cdots b_2 b_1$ where two vertices are adjacent if and only if their binary strings differ by a single bit. An *induced path* in an undirected graph $G$ is a path where every pair of non-consecutive vertices in the path are non-adjacent in $G$. Induced paths are sometimes called *snakes* and the problem of finding the longest snakes in $Q_n$ is known as the *snake-in-the-box* problem.

A generalization of this problem is to find the longest $k$-snake in $Q_n$. A *$k$-snake* is defined to be a path in a graph $G$ such that every pair of vertices at distance at least $k$ in the path are also at distance at least $k$ in $G$. Thus a 1-snake is a simple path and a 2-snake is an induced path. Similar problems and definitions apply for cycles. The *coil-in-the-box* problem is the problem of finding the longest induced cycles in $Q_n$. The more general problem is to find the optimal *$k$-coils* in $Q_n$. A *$k$-coil* is defined to be a cycle in a graph $G$ such that every pair of vertices at distance at least $k$ in the cycle are also at distance at least $k$ in $G$. A 1-coil is a simple cycle and a 2-coil is an induced cycle. The parameter $k$ is referred to as the *spread*.

The problem of finding long $k$-coils in $Q_n$ was first considered by Singleton [36] using explicit constructions. Subsequently, Klee [26,25] studied the topic using the term *circuit codes* in place of $k$-coils. Since then, the lower bounds for $k$-coils have been improved by Deimer [10], Paterson and Tuliani [31], Hiltgen and Paterson [20],

School of Computer Science, University of Guelph, Canada
E-mail: simon.hood@gmail.com, drecoski@uoguelph.ca, jsawada@uoguelph.ca, cwong@uoguelph.ca

| $n$ | $k$ | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 6* | 6* | 6* | 6* | 6* | 6* |
| 4 | 8* | 8* | 8* | 8* | 8* | 8* |
| 5 | 14* | 10* | 10* | 10 * | 10* | 10* |
| 6 | 26* | 16* | 12* | 12* | 12* | 12* |
| 7 | 48* | 24* | 14* | 14* | 14* | 14* |
| 8 | 96 | 36* | 22* | 16* | 16* | 16* |
| 9 | 188 | 64[a] (58) | 30* | 24* | 18* | 18* |
| 10 | 348 | 102[a] (100) | 46** | 28* | 20* | 20* |
| 11 | 640 | 160 | 70[a] (68) | 40** | 30* | 22* |
| 12 | 1238 | 288 | 102[a] (98) | 60[a] (58) | 36** | 32* |
| 13 | 2468 | 494[b] (442) | 182 | 80[a] (78) | 50** | 36* |
| 14 | 4934 | 812[b] (700) | 280 | 106[a] (102) | 68[a] (66) | 48** |
| 15 | 9868 | 1380[b] (1290) | 480[b] (450) | 210 | 88[a] (82) | 60[a] (58) |
| 16 | 19740 | 2240[b] (2176) | 768[b] (672) | 288 | 118[a] (106) | 76[a] (72) |
| 17 | 39840 | 3910[b] (3842) | 1224[b] (1088) | 476 | 204 | 102[a] (90) |

* value previously known to be optimal
** newly verified to be optimal
[a] found by partial exhaustive search
[b] found by optimized implementation of construction from [31]

**Table 1** Longest known $k$-coils. The numbers in parentheses represent best previously known values.

Zinvoik et. al [49], and Chebiryak and Kroening [7]. In [20], applications for $k$-coils (circuit codes) are clearly described. To the best of our knowledge, the $n$-dimensional hypercubes are the only class of graphs for which the $k$-snake and $k$-coil problems have been studied.

In this paper we present two algorithms that exhaustively search for long $k$-snakes and $k$-coils in $Q_n$. The first is an adaptation of Kochut's [28] algorithm for $k$=2 which requires several non-trivial changes for general $k$, especially in the case of $k$-coils. At each recursive step it requires $O(n^{k-1})$ time to update data structures. The second algorithm requires $O(t)$ time at each recursive call, where $t$ is the current length of the $k$-snake being searched. The latter algorithm is much more efficient for some small values of $n$ and larger values of $k$. For $k$-coils, we consider two optimization strategies: the first considers a connectivity constraint and the second takes advantage of rotational equivalence. For rotational equivalence, several heuristics are considered with varying amounts of overhead. Using the best of our optimization strategies we were able to find 13 new longest $k$-coils and 21 new longest $k$-snakes. We also were able to verify that a number of previously known longest $k$-coils and $k$-snakes are indeed optimal. Since many heuristic based search algorithms in some way depend on an exhaustive search, the ideas here may provide improvements to previously studied heuristic algorithms [41,5,6,12,33,3,24, 4].

In addition to exhaustive search, we also applied some optimizations to a *single-track circuit code* construction algorithm by Paterson and Tuliani [31]. This allowed us to find an additional 8 longest known $k$-coils. Using these longest $k$-coils, we applied some basic backtracking to find 18 new longest $k$-snakes. Table 1 provides the lengths of the longest known $k$-coils and Table 2 provides the lengths for the longest known $k$-snakes. Since there has been less reporting on $k$-snakes, many of the previously best known bounds are obtained by removing $k-1$ consecutive nodes from a longest corresponding $k$-coil. This results in a reduction of $k$ in the length of the $k$-snake compared to the $k$-coil. The previous best known results were accumulated from [31,49,7,32,24,45].

The remainder of the paper is outlined as follows. In Section 2 we outline two search algorithms with a comparative analysis. In Section 3 we outline a variety of optimizations for $k$-coils and provide an experimental analysis. In Section 4 we introduce single-track circuit codes and include an updated table of the longest known such codes. A summary is provided in Section 5. Instances of our new longest $k$-coils and $k$-snakes are provided in the Appendix. All of our searches and experiments were performed using SHARCNET[1].

[1] This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca) and Compute/Calcul Canada.

| $n$ | $k$ | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 4* | 3* | 3* | 3* | 3* | 3* |
| 4 | 7* | 5* | 4* | 4* | 4* | 4* |
| 5 | 13* | 7* | 6* | 5* | 5* | 5* |
| 6 | 26* | 13* | 8* | 7* | 6* | 6* |
| 7 | 50* | 21* | 11* | 9* | 8* | 7* |
| 8 | 98 | 35** (33) | 19* | 11* | 10* | 9* |
| 9 | 190 | 63[a] (53) | 28** (26) | 19* | 12* | 11* |
| 10 | 370 | 103[b] (83) | 47** (42) | 25** (23) | 15* | 13* |
| 11 | 695 | 157[b] (151) | 68[a] (64) | 39** (35) | 25** (24) | 15* |
| 12 | 1274 | 286[b] (285) | 104[a] (94) | 56[a] (51) | 33** (30) | 25* |
| 13 | 2466 | 493[b] (439) | 181[b] (178) | 79[a] (73) | 47[a] (44) | 31** (29) |
| 14 | 4932 | 811[b] (697) | 279[b] (276) | 112[a] (97) | 66[a] (60) | 42[a] (41) |
| 15 | 9866 | 1379[b] (1287) | 480[b] (446) | 206[b] (205) | 89[a] (76) | 55[a] (51) |
| 16 | 19738 | 2240[b] (2173) | 766[b] (668) | 285[b] (283) | 117[a] (100) | 72[a] (65) |
| 17 | 39838 | 3941[b] (3839) | 1223[b] (1084) | 473[b] (471) | 200[b] (198) | 98[b] (83) |

* value previously known to be optimal
** newly verified to be optimal
[a] found by partial exhaustive search
[b] found by starting from a longest known coil and backtracking

**Table 2** Longest known $k$-snakes. The numbers in parentheses represent best previously known values.

## 2 Exhaustive search algorithms

Kochut's algorithm [28] that exhaustively searches for optimal 2-snakes follows a straightforward recursive backtracking approach with one key optimization: it takes advantage of the symmetry of $Q_n$ to assume that every snake starts at $0^n$ and each dimension $d$ is visited before any dimension greater than $d$ (i.e., it considers equivalence under permutation of the bit positions). Thus, the following 3 snakes are equivalent with the first being the canonical form searched by the algorithm:

[0000, 0001, 0011, 0111, 0110], [0000, 1000, 1010, 1110, 0110], [1111, 0111, 0101, 0001, 1001].

In the next two subsections we outline two approaches that apply this optimization. In the final subsection we provide a comparative analysis. To simplify our discussion, we say that a vertex $y$ *conflicts* with vertex $x$ if the two vertices differ in fewer than $k$ bit positions.

### 2.1 Approach one: $\mathsf{Search}(t, d)$

The first search method we discuss follows the standard exhaustive search approach used for $k = 2$. The basic idea is to update the conflicts with unused vertices as the search path gets extended. This allows a constant time test to determine whether or not a search path can be extended through a given vertex $x$. The data structure required to maintain the conflicts is an array *numConflicts* where each *numConflicts*$[x]$ stores the number of vertices in the current search path $\alpha$ that are in conflict with $x$. Since each vertex is in conflict with $c = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{k-1}$ other vertices in $Q_n$, updating *numConflicts* will require $O(c) = O(n^{k-1})$ time every time a vertex is added to the path.

Pseudocode for such a recursive $k$-snake and $k$-coil search algorithm in $Q_n$ is illustrated by the function $\mathsf{Search}(t, d)$ in Figure 1. The $k$-snake ($k$-coil) being searched at the start of each recursive call is stored in $\alpha = a_0 \cdots a_{t-1}$, where each $a_i$ is the integer representation of vertex $i$. The parameter $t$ indicates the length of the current path and $d$ indicates the largest dimension currently visited. The other global variables used are:

- *numConflicts*$[x]$: the number of vertices in $\alpha$ that conflict with $x$,
- $adj[x][d]$: the neighbor of $x$ obtained by flipping the $d$-th bit,
- *conflict*$[x]$: the set of $c$ vertices in $Q_n$ that conflict with $x$,

Note that the arrays $adj$ and *conflict* are easily pre-computed. To run the algorithm, we initialize *numConflicts*$[x] = 0$ for each vertex $x$ and then set the first $k+1$ vertices in the path to $0, 2^1-1, \ldots, 2^k-1$, incrementing

**procedure** Search($t, d$: **int**)
**int** $i, x, y$
**for** $i$ **from** 1 **to** Min($d+1, n$) **do**
  $x := a_t := adj[a_{t-1}][i]$
  **if** $numConflicts[x] < k$ **then**
    **for each** $y \in conflict[x]$ **do** $numConflicts[y]{+}{+}$
    CheckMaximal($t$)
    Search($t+1$, Max($i, d$))
    **for each** $y \in conflict[x]$ **do** $numConflicts[y]{-}{-}$

**Fig. 1** Exhaustive search algorithm for optimal $k$-snakes or $k$-coils in $Q_n$.

**function** CheckMaximal($t$: **int**) **returns boolean**
**int** $x$
**if** $bits[a_t] > k$ **or** $maxlen \geq t + k$ **then return** FALSE
**if** $bits[a_t] = 1$ **then**
  Print($a_0 \cdots a_t$)
  $maxlen := t + 1$
  **return** TRUE
**for each** $x \in adjSmaller[a_t]$ **do**
  $a_{t+1} := x$
  **if** $numConflicts[x] = k$ **and** CheckMaximal($t+1$) **then return** TRUE
**return** FALSE

**Fig. 2** Recursive function to check if an initial search path $a_0 \cdots a_t$ can be extended to a $k$-coil.

the conflicts for each vertex. The initial recursive call is Search($k+1, k$). During a recursive call, each neighbor $x$ of the last vertex $a_{t-1}$ is checked to see if it extends the path without violating the constraints on $k$ or $d$. In order to abide by the constraint on $k$, there can be at most $k-1$ vertices (the last $k-1$ vertices) that conflict with $x$. If $x$ does not violate the constraints, we increment *numConflicts*$[y]$ for each vertex $y$ that conflicts with $x$ and then recursively look for longer $k$-snakes.

If we are interested in generating $k$-snakes, the function CheckMaximal($t$) will check if $a_0 \cdots a_t$ is the longest $k$-snake found so far. By maintaining the variable $maxlen$ to represent the longest $k$-snake found in the search, a new longest $k$-snake will be found if $maxlen < t$. For $k$-coils, the test is more complicated because the search algorithm will never visit vertices with fewer than $k$ bits after the initialization steps. Thus when $a_t$ has exactly $k$ bits set to 1 and $maxlen < t + k$, we perform a special test to see if there is a valid direct path from $a_t$ to $a_0$.

To efficiently perform the $k$-coil test, we pre-compute two additional data structures. First, we let $bits[x]$ store the number of bits set to 1 in the binary representation of $x$. Second, we let $adjSmaller[x]$ be a set of the $bits[x]$ vertices that are neighbors of $x$ and have one fewer bit set to 1. Using these data structures, the function CheckMaximal($t$) shown in Figure 2 recursively checks if a direct path exists back to the starting vertex $a_0$. Observe that for each recursive call, each neighbor of $a_t$ with one fewer bit set to 1 can be visited efficiently using $adjSmaller[a_t]$, extending the $k$-coil one step closer to the start vertex 0. Note that each vertex in the return path will conflict with exactly $k$ vertices in the original search path $a_0 \cdots a_t$ being tested. For instance, if $k = 4$ the vertex $a_{t+1}$ (in the original recursive call) will have $k-1$ bits set to 1. It will be distance 1 from $a_{t-1}$, distance 2 from $a_{t-2}$, distance 3 from $a_{t-3}$ and also distance 3 from $a_0$, but a distance of at least $k$ from all other vertices in $\alpha$. In the base case when the last vertex is adjacent to 0 (it will have one bit set to 1), we update $maxlen$ and print out the $k$-coil.

In the worst case, if no such path is found, the function CheckMaximal($t$) for $k$-coils will run independently of $n$ in $O(k!)$ time. However, since the recursive check is rarely required, the running time of this function has little impact on the overall search time. As an example, in the case of $n = 6$ and $k = 2$ there are a total of 651075 recursive search calls, but the $O(k!)$ recursive check is only performed 22 times. Since CheckMaximal($t$) is very efficient for both $k$-snakes and $k$-coils, the $O(c) = O(n^{k-1})$ update of the conflicts dominates the running time of Search($t, d$).

Note that the space required by this algorithm is $O(n^{k-1}2^n)$. This is the space needed to store the array $conflict$. This can be reduced to $O(2^n)$ if both the conflicts and the adjacencies are computed on demand. Each of these operations can be done efficiently using bitwise operations on integers.

**function** IsConflict($t, s$: **int**) **returns boolean**
**int** $j$
**for** $j$ **from** $t-k-1$ **down to** $s$ **do**
    **if** *inConflict*$[a[t]][a[j]]$ **then return** TRUE
**return** FALSE

**procedure** Search2($t, d$: **int**)
**int** $i$
**for** $i$ **from** 1 **to** Min($d+1, n$) **do**
    $a_t := adj[a_{t-1}][i]$
    **if not** IsConflict($t$, 0) **then**
        CheckMaximal($t$)
        Search2($t+1$, Max($i, d$))

**Fig. 3** Alternate search algorithm for $k$-snakes and $k$-coils in $Q_n$.

### 2.2 Approach two: Search2($t, d$)

For $k = 2$ the time required to update conflicts in Search($t, d$) is an efficient $O(n)$, however as $k$ gets bigger the overhead for the updates can be significantly larger than the length of a optimal $k$-snake or $k$-coil. For instance, when $n = 10$ and $k = 5$ each vertex has 385 conflicts. Thus adding a vertex to a search path requires executing two **for** loops iterating 385 times each. The updates are performed so it is possible to test whether a vertex $x$ can extend the current search path in constant time. As an alternative, the following method to check conflicts will be more efficient in some cases: test $x$ for a conflict with each vertex in the current search path not including the last $k-1$ vertices. In fact, when considering adding $x = a_t$ to the search path $a_0 \cdots a_{t-1}$, we need only see if $x$ is in conflict with any of the vertices in $a_0 \cdots a_{t-k-1}$, since if $x$ is in conflict with $a_{t-k}$, it will also be in conflict with $a_{t-k-1}$. During each recursive call, testing for these conflicts must be done for each of the $n$ neighbors of $a_{t-1}$. Observe that exactly $k$ of these neighbors will be in conflict with $a_{t-k-1}$. The remaining $n-k$ neighbors that are not in conflict with $a_{t-k-1}$ will require $t-k$ comparisons to test for conflicts in the worst case. Therefore, using this alternate approach, in the worse case each recursive call will require $(n-k)(t-k)+k$ comparisons. When $n = 10$ and $k = 5$, the longest $k$-snake is 25, so the worst case will only require $5 \cdot 20 + 5 = 105$ comparisons. This is a significant improvement over the 770 loop iterations required by Search($t, d$).

Pseudocode for this alternate approach, Search2($t, d$) is given in Figure 3. It requires the same initialization as Search($t, d$). The function IsConflict($t, s$) tests whether or not $a_t$ is in conflict with any vertex in $a_{t-k-1} \cdots a_s$. It assumes that the $conflict[x][y]$ is precomputed to store a 1 if the $x$ conflicts with $y$ and store a 0 otherwise. Because the array *numConflicts* is no longer maintained, the function CheckMaximal($t$) requires a more expensive test for conflicts as well. Specifically, a vertex will be in conflict if IsConflict($t+1, k - bits[a[t]] + 1$) returns TRUE. However, since this test is so rarely required, it will have an insignificant impact on the search time.

Note that the space required by this algorithm is $O(4^n)$ to store the values in the 2-dimensional array *inConflict*. If testing for a conflict between two vertices is done on demand and the adjacencies are also computed on demand (each of these operations can be done efficiently using bitwise operations on integers), then the space requirement can be reduced to $O(s)$ where $s$ is a bound on the maximum length of the snake or coil that can be generated.

### 2.3 Comparing algorithms

We compare the running times (measured in seconds) of the two approaches for some values of $n$ and $k$ in the following table.

| $(n, k)$ | (7,2) | (8,3) | (9,4) | (10,5) | (11,5) | (12,6) |
|---|---|---|---|---|---|---|
| **Search** | 680210 | 137 | 5.04 | 1.02 | 23606 | 2009 |
| **Search2** | 2334000 | 214 | 2.93 | 0.26 | 5464 | 168 |

While difficult to prove, it is reasonable to assume that for each $k$, there is an integer $m$ such that for all $n < m$, Search2 will be faster than Search, and that for all $n \geq m$, the first approach will be faster. To illustrate this further, we compare the number of updates to the array *numConflicts* required to generate each recursive call to

Search with the worst case number of comparisons required to determine conflicts during each recursive call to Search2. Recall that these two values are represented by $2c = 2(\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{k-1})$ and $(n-k)(s-k)+k$, where $s$ represents the longest known $k$-snake of length $n$.

| $(n,k)$ | (6,3) | (7,3) | (8,3) | (10,4) | (11,4) | (12,4) | (12,5) | (13,5) | (14,5) |
|---|---|---|---|---|---|---|---|---|---|
| $2c$ | 42 | 56 | 72 | 350 | 462 | 596 | 1586 | 1984 | 2540 |
| $(n-k)(s-k)+k$ | 33 | 75 | 163 | 262 | 452 | 804 | 362 | 597 | 914 |

For $k = 2$, Search always has fewer comparisons than Search2 (this is easy to verify). For $k = 3$, Search2 has fewer comparisons when $n \leq 6$. For $k = 4$, Search2 has fewer comparisons for all $n < 12$. For $n = 12$, it is hard to evaluate because even though $804 > 596$, the value for 804 is worst case and not average case. For $k = 5$ it is clear that Search2 has fewer comparisons for $n \leq 14$.

## 3 Optimizations for $k$-coils

In the next two subsections, two approaches that attempt to optimize the search algorithm for $k$-coils will be discussed. The first approach considers connectivity and the second approach considers rotational equivalence. Recall that the number of bits set to 1 for each vertex $v$ is pre-computed in $bits[v]$.

### 3.1 Connectivity

In this subsection we consider a simple connectivity property when searching for optimal $k$-coils. In particular, given a $k$-snake $\alpha = a_0 \cdots a_t$ in $Q_n$, if we know that it is impossible to extend $\alpha$ into a $k$-coil, then we can terminate this search branch of the computation tree. Unfortunately, a complete test for connectivity does not seem practical since even an $O(2^n)$ breadth-first-search is not sufficient. Thus, we discuss a simplified but efficient test that can be used to detect some of these dead ends.

Consider a partition of the vertices into $n+1$ subsets $V_0 \cdots V_n$ such that each subset $V_i$ contains the $\binom{n}{i}$ bitstrings with $i$ bits set to 1. The basic idea of our approach is as follows: if a vertex from $V_i$ is appended to the current $k$-snake so that for some $j < i$ no vertex in $V_j$ can be reached without violating the conflict constraint, then it is impossible to return to $a_0 = 0$. For example, consider the $k$-snake [0, 1, 3, 7, 6, 14, 12, 28, 24, 56, 48, 52] for $n = 6$ and $k = 2$ using integer representations for the vertices. The subset $V_1 = \{1, 2, 4, 8, 16, 32\}$ has no available vertices since 3 is adjacent to 1 and 2, 12 is adjacent to 4 and 8, and 48 is adjacent to 16 and 32. Thus using the partial connectivity test we can terminate this search branch of a $k$-snake at length 11, which is considerably shorter than the optimal $k$-coil of length 26.

It is possible to efficiently implement this optimization in $O(n^{k-1} + k^2)$ time in the worst case which is comparable to the time required to update the array *numConflicts* using Search$(t, d)$. Thus the extra work only increases the complexity by a constant factor per recursive call. However, because the extra work almost doubles the computation required at each recursive call, it is important to reduce the number of recursive calls by a significant factor for this optimization to be useful. We illustrate the effect of this optimization with respect to the number of recursive calls required for some small values of $n$ and $k$ in the following table:

| $(n,k)$ | (6,2) | (7,3) | (8,3) | (8,4) | (9,4) | (9,5) | (10,5) |
|---|---|---|---|---|---|---|---|
| **Non-optimized** | 651075 | 26060 | 556120186 | 4624 | 7817954 | 1963 | 557993 |
| **Connectivity** | 389257 | 15719 | 300717441 | 2424 | 3840003 | 1345 | 277788 |
| **% Reduction** | 40 | 40 | 46 | 48 | 51 | 31 | 50 |

It does not appear that this optimization leads to significant savings in run-time. In fact, for each of the experiments, the optimized version actually requires slightly more time. One reason why this heuristic does not appear to be effective may be because most of the dead ends found occur deep in the recursion.

### 3.2 Rotational equivalence

Since $k$-coils are cycles it is natural to consider equivalence under rotation. For example, consider the 2-coil of length 8 illustrated in Figure 4. Depending on which of the 8 vertices is the starting vertex, up to 8 different

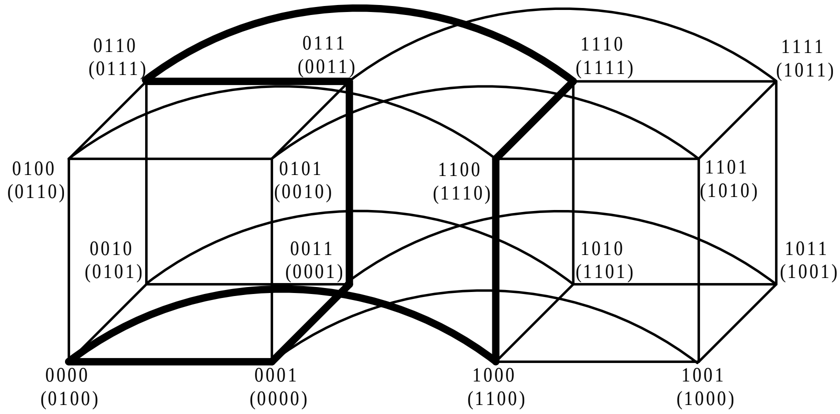


**Fig. 4** Illustration of two rotationally equivalent 2-coil sequences in $Q_4$.

equivalent vertex sequences can be obtained, even when the vertices are relabled so they match the canonical form outlined earlier: label the first vertex $0^n$ and visit dimension $d$ before $d+1$. The two different, but equivalent 2-coils illustrated in this figure are:

$$[0000, 0001, 0011, 0111, 0110, 1110, 1100, 1000],$$
$$[0000, 0001, 0011, 0111, 1111, 1110, 1100, 0100].$$

In this subsection we explore four different methods for testing for partial rotational equivalence. For a given rotational test, we say a search path $\alpha$ is a *dead end* if it is not in a specified canonical form. We use the term *canonical* loosely, since the tests are only partial tests and equivalent $k$-coils may be generated from different canonical branches. In one of the tests, we also take advantage of equivalence under reversal. However, in general, this is an expensive test for at best an additional 50% reduction in the search space.

We visit the four methods in order of time complexity per recursive call to perform the rotational test. Each optimization is independent of which search algorithm it is applied to: Search or Search2. They can be summarized as follows:

- ▷ diagonal $O(1)$,
- ▷ pre-diagonal $O(n)$,
- ▷ pre-diagonal w/reversal $O(n)$,
- ▷ bit count sequence $O(t) = O(snakelength)$.

In Section 3.2.5, we provide an analysis comparing the impact on the number of recursive calls and the overall run time for feasible values of $n$ and $k$.

### *3.2.1 The diagonal*

The first method can be implemented very simply and efficiently. We say that two vertices are *diagonal* to each other in $Q_n$ if the two vertices are complements to each other, i.e., their integer representations sum to $2^n - 1$. If two vertices $a_i$ and $a_j$ are diagonal to each other in a search path $\alpha$ where $i < j$, then the *length* of the diagonal is given by $j - i$; if one of the vertices is not in the path then the length of their diagonal is $2^n$. If we let $len$ denote the length of the diagonal between $a_0 = 0$ and its complement, then we say a search path $\alpha$ is in a canonical form if every diagonal has length at least $len$. For example when $n = 6$ and $k = 2$, consider the following search path $\alpha$ where the vertices are represented by integers: [0, 1, 3, 7, 6, 14, 12, 13, 29, 31, 63, 55, 51]. Since $len = 10$ and the diagonal between 12 and 51 has length 6, this path is not in a canonical form. Thus $\alpha$ is a dead end.

The following steps can be used to efficiently implement this optimization:

1. Maintain the global variable $len$ to store the length of the diagonal between $0^n$ and $1^n$.
2. Maintain the global array $pos$ where $pos[v]$ is the index of the vertex $v$ in $\alpha$ (or $-2^n$ if not in $\alpha$).
3. Detect a dead end if $t - pos[2^n{-}1{-}a_t] < len$.

Observe that these modifications can easily be applied in $O(1)$ time per recursive call.

*3.2.2 The pre-diagonal*

In an attempt to remove more isomorphic branches of the exhaustive search, we consider a slightly more expensive test for equivalence. A vertex $x$ is said to be *pre-diagonal* to a vertex $y$ if the two vertices differ in $n-1$ bit positions. Thus each vertex will have $n$ vertices that are pre-diagonal to it. For example, the 5 pre-diagonals to 00000 are 01111, 10111, 11011, 11101, 11110. If $a_i$ and $a_j$ are pre-diagonal to each other in a path $\alpha$ where $i < j$, then the length of the pre-diagonal is $j - i$. Now, similar to our diagonal test, by focusing on the length of all pre-diagonals in a search path we can obtain a canonical form. If $len$ denotes the length of the shortest pre-diagonal including $0^n$, then we say a search path $\alpha$ is in a canonical form if every pre-diagonal has length at least $len$. If there is no pre-diagonal with $0^n$, then $len = 2^n$. For example, if $n = 6$ and $k = 2$, the search path [0, 1, 3, 7, 15, 13, 29, 61] is a dead end since the shortest pre-diagonal with 0 is 61 (111101) at length 7 while the length between 3 (000011) and 61 (111101) is 5. Similarly, the search path [0, 1, 3, 7, 6, 14, 30] is a dead end since 1 (000001) and 30 (011110) are pre-diagonals but there is no pre-diagonal with 0.

The following steps can be used to efficiently implement this optimization:

1. Pre-compute the set $pdlist[v]$ for each vertex $v$ that contains the $n$ vertices that are pre-diagonal to $v$.
2. Maintain the global variable $len$ to store the length of the shortest pre-diagonal with $0^n$.
3. Maintain the global array $pos$ where $pos[v]$ is the index of the vertex $v$ in $\alpha$ (or $-2^n$ if not in $\alpha$).
4. Detect a dead end if $t - pos[v] < len$ for any $v \in pdlist[a_t]$.

Detecting a dead end requires $O(n)$ time in the worst case and the other variables are easily maintained in $O(1)$ time. Thus, these modifications can easily be applied in $O(n)$ time per recursive call.

*3.2.3 Refined pre-diagonal with reversal*

The pre-diagonal test still allows equivalent search paths where there are multiple pre-diagonals of length $len$. To remove even more equivalent branches of computation we refine our definition of a canonical search path by considering the second longest pre-diagonals from each vertex. Let $len2$ denote the second longest pre-diagonal containing 0. If no such pre-diagonal exists then $len2 = 2^n$. A search path is in a canonical form if (1) every pre-diagonal has length at least $len$ and (2) if a vertex has a pre-diagonal of length $len$ then the length of its second longest pre-diagonal must be at least $len2$. For example, consider the following search path where $n = 6$ and $k = 3$: [0, 1, 3, 7, 15, 31, 30, 28, 60]. The only pre-diagonal with 0 is 31 with $len = 5$, but 1 is pre-diagonal with both 30 and 60. The lengths of these pre-diagonals are 5 and 7 respectively, so this search path is a dead end.

Since $k$-coils also have equivalence under reversal, we can refine our canonical form even further by enforcing this condition on the reversal. As an example with $n = 6$ and $k = 3$, consider [0, 1, 3, 7, 15, 31, 29, 28, 60]. Again, the only pre-diagonal with 0 is 31 with $len = 5$, however 60 is pre-diagonal with both 7 and 1. The lengths in the reverse direction from 60 are 5 and 7 respectively, so this path is a dead end.

In addition to the modifications described in the previous subsection, the following steps can be used to efficiently implement this optimization:

1. Maintain the global variable $len2$ in $O(1)$ time.
2. Maintain the boolean array $b$ such that $b[v]$ is TRUE if and only if the vertex $v$ is in the search path and has a pre-diagonal at length $len$.
3. Detect a dead end if for any $v \in pdlist[a_t]$ where $b[v] =$ TRUE we have $t - pos[v] < len2$.
4. Compute two boolean flags: $f_1$ and $f_2$. The flag $f_1$ is TRUE if and only if there exists a $v \in pdlist[a_t]$ such that $t - pos[v] = len$. The flag $f_2$ is TRUE if and only if there exists a $v \in pdlist[a_t]$ such that $len < t - pos[v] < len2$.
5. Detect a dead end (for reversals) if both $f_1$ and $f_2$ are TRUE.

Maintaining the array $b$ and setting the flags can easily be done in $O(n)$ time. Thus, for this heuristic detecting a dead end requires at most $O(n)$ time.

### 3.2.4 Bit count sequence

A more complete test for rotational equivalence applies a lexicographic approach to the path being searched. The bit count sequence of a search path $\alpha = a_0 \cdots a_t$ is a sequence of the number of bits in each vertex $a_i$ that differ from $a_0$. For example, recall the two equivalent 2-coils introduced earlier:

$$[0000, 0001, 0011, 0111, 0110, 1110, 1100, 1000],$$
$$[0000, 0001, 0011, 0111, 1111, 1110, 1100, 0100].$$

The bit count sequences for these 2-coils are $\langle 0,1,2,3,2,3,2,1\rangle$ and $\langle 0,1,2,3,4,3,2,1\rangle$ respectively. We say a search path is in canonical form if the bit count sequence starting from 0 is lexicographically largest compared to the bit count sequences starting from the other vertices in the search path. As an example, consider the search path $\alpha$ illustrated in Figure 5. The bit count sequence is $\langle 0,1,2,3,4,3,4,3,4,3,2\rangle$. However, the bit count sequence relative to starting at vertex 01101 is $\langle 0,1,2,3,4,5\rangle$. Since this is lexicographically larger than the bit count sequence for $\alpha$, the search path $\alpha$ is not in canonical form and we detect a dead end.

| $\alpha$ | 00000 | 00001 | 00011 | 00111 | 01111 | 01101 | 11101 | 11001 | 11011 | 11010 | 10010 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000 | **0** | **1** | **2** | **3** | **4** | **3** | **4** | **3** | **4** | **3** | **2** |
| 00001 | | 0 | 1 | 2 | 3 | 2 | | | | | |
| 00011 | | | 0 | 1 | 2 | 3 | 4 | 3 | 2 | | |
| 00111 | | | | 0 | 1 | 2 | 3 | 4 | 3 | 4 | 3 |
| 01111 | | | | | 0 | 1 | 2 | 3 | 2 | | |
| **01101** | | | | | | **0** | **1** | **2** | **3** | **4** | **5** |
| 11101 | | | | | | | 0 | 1 | 2 | 3 | 4 |
| 11001 | | | | | | | | 0 | 1 | 2 | 3 |
| 11011 | | | | | | | | | 0 | 1 | 2 |
| 11010 | | | | | | | | | | 0 | 1 |
| 10010 | | | | | | | | | | | 0 |

**Fig. 5** A dead end search path $\alpha$ when considering rotational equivalence using the bit count sequence.

Note that once a bit-count sequence is smaller than the bit-count sequence of $\alpha$, like the one starting with 00001, $\langle 0,1,2,3,2\rangle$, we no longer need to consider it for rotational testing. Also, even though they are illustrated in the table, we do not need to consider the bit count sequences starting from $a_t, a_{t-1}, \ldots, a_{t-k}$, as they will always start $0, 1, \ldots, k$.

The following steps can be used to efficiently implement this optimization:

1. Pre-compute the Hamming distance between every pair of vertices $u$ and $v$ in *diff*$[u][v]$.
2. Maintain the global array $test$, initialized to $2^n$, to determine which starting positions require testing. When appending $a_t$, if the bit-count sequence starting from $a_i$ becomes less than the bit-count sequence for $\alpha$, then $test[i]$ gets updated to $t$. If $test[i] < t$, the bit-count sequence starting from $a_i$ can be safely ignored.

Using the pre-computed array *diff* it is possible to test each bit-count sequence in constant time: the next value in the bit-count sequence relative to $a_i$ is *diff*$[a_i][a_t]$. Thus, this optimization requires $O(t-k)$ time.

A slight improvement can be made if we maintain a linked list that contains only the positions $i$ that need to be tested. To maintain the linked list requires extra overhead (and code-complexity) as elements have to be removed during the test and then restored after the recursive call. However, in the amortized sense, it requires only a constant amount of extra work for each vertex in the list. In the following subsection we report experimental results with and without using such a linked list.

### 3.2.5 Experimental results

In Table 3, the number of recursive calls required by the non-optimized search algorithm is compared to the four optimized approaches utilizing rotational equivalence is considered. Generally, the more expensive the optimization, the larger the reduction in the number of recursive calls.

| $(n,k)$ | (6,2) | (7,3) | (8,3) | (8,4) | (9,4) | (9,5) | (10,5) |
|---|---|---|---|---|---|---|---|
| **Non-optimized** | 651075 | 26060 | 556120186 | 4624 | 7817954 | 1963 | 557993 |
| **Diagonal** | 114042 | 13940 | 365822530 | 2329 | 6647281 | 898 | 442018 |
| **Pre-diagonal** | 226152 | 8299 | 85528048 | 1359 | 952904 | 682 | 92333 |
| ▷ **Pre-diagonal w/reversal** | 85643 | 3415 | 47446549 | 802 | 617453 | 498 | 77030 |
| **Bit count** | 50337 | 2647 | 26620597 | 542 | 432632 | 372 | 27611 |

**Table 3** A comparison on the number of recursive calls required for exhaustive search on select $k$-coils.

It is interesting to note the impact on the number of recursive calls in case additional processing is desired by some application as each new vertex is appended to a search path. But ultimately we are interested in evaluating the impact that each optimization has on the overall running time. Such timing results are given in Table 4. The base search algorithm Search was used for ($n = 7, k = 2$) and ($n = 8, k = 3$), while Search2 was more efficient for the remaining cases. Each experiment was run on an Opteron 2.2 GHz processor. In each case, the bit count sequence implemented with a linked list was the most efficient.

| $(n,k)$ | (7,2) | (8,3) | (10,4) | (11,5) | (12,6) |
|---|---|---|---|---|---|
| **Non-optimized** | 680210 | 137 | 1303000 | 5460 | 168 |
| **Diagonal** | 46641 | 92 | 1351000 | 6510 | 181 |
| **Pre-diagonal** | 128652 | 24 | 199800 | 2590 | 134 |
| ▷ **Pre-diagonal w/reversal** | 77660 | 16 | 197700 | 2415 | 143 |
| **Bit count** | 45756 | 9 | 60800 | 255 | 7 |
| ▷ **Bit count w/linked list** | 35652 | 8 | 58000 | 243 | 7 |

**Table 4** A comparison on the running time (in seconds) required to exhaustively search some $k$-coils for various values of $n$. Search was used for (7,2) and (8,3) and Search2 was used for (10,4), (11,5), (12,6).

### 3.3 Applying the optimized algorithms

Using the optimized exhaustive search algorithm, we used SHARCNET to search for new long $k$-snakes and $k$-coils. Using 10 processors to perform the search with a limit of 7 days for each $(n, k)$ pair, we were able to find 13 new longest $k$-coils, 21 new longest $k$-snakes. We also verified that several previously known $k$-snakes and $k$-coils were indeed optimal. These results are shown in Table 1 and Table 2. Instances of each new longest $k$-coil and $k$-snake are given in the Appendix.

## 4 Single track circuit codes

A *single-track circuit code* is a $k$-coil (circuit code) with an additional property: each track (the sequence of bits obtained by isolating a given bit position) is a cyclic shift of the first track. For example, consider the the two 2-coils below.

| | |
|---|---|
| 0000 | 0000 |
| 0001 | 0001 |
| 0011 | 0011 |
| 0111 | 0111 |
| 1111 | 1111 |
| 1110 | 1011 |
| 1100 | 1010 |
| 1000 | 1000 |

The first track in the left 2-coil is the sequence of the bits from the left most bit position: 00001111. The 2nd track is 00011110, the third track is 00111100, and the 4th track is 01111000. Each of these tracks is a cyclic shift of

| $n$ | $k$ | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 4* | 4* | - | - | - | - |
| 3 | 6* | 6* | 6* | - | - | - |
| 4 | 8* | 8* | 8* | 8* | - | - |
| 5 | 30* | 10* | 10* | 10* | 10* | - |
| 6 | 60* | 24* | 12* | 12* | 12* | 12* |
| 7 | 126* | 42* | 14* | 14* | 14* | 14* |
| 8 | 240 | 80 | 16 | 16* | 16* | 16* |
| 9 | 504* | 162 | 54 | 18 | 18* | 18* |
| 10 | 960 | 320 | 80 | 20 | 20* | 20* |
| 11 | 2046* | 594 | 154 | 22 | 22 | 22* |
| 12 | 3960 | 960 | 288 | 96 | 24 | 24 |
| 13 | 8190* | 1898 | 494 (442) | 182 | 26 | 26 |
| 14 | 16128 | 3528 | 812 (700) | 280 | 28 | 28 |
| 15 | 32730 | 6630 | 1380 (1290) | 480 (450) | 210 | 30 |
| 16 | 65504 | 12512 | 2240 (2176) | 768 (672) | 288 | 32 |
| 17 | 131070* | 22406 | 3910 (3842) | 1224 (1088) | 476 | 204 |

* value previously known to be optimal

**Table 5** Longest known single-track circuit codes. The numbers in parentheses represent best known values previous to this research.

the first track and so the 2-coil is also a single-track circuit code. As another example, consider the 2-coil on the right. The first track is 00001111 and the second track is 00011000. Since the second track is not a cyclic shift of the first track the 2-coil does not have the single-track property. The notion of the single-track property was first introduced in [21] with a focus on circuit codes in [20].

Many of the previously known longest $k$-coils are due to a construction for single-track circuit codes by Paterson and Tuliani [31]. By implementing their construction with an optimization that considers equivalence under rotation, we were able to extend many of their previously reported results. In particular, we found 8 new longest single-track circuit codes for various $n$ and $k$ as illustrated in Table 5. Instances of these new longest single-track circuit codes are listed in the Appendix and they also represent 8 new longest $k$-coils. By applying backtracking to these long $k$-coils, we have also found many new longest known $k$-snakes. These new bounds are shown in Table 1 and Table 2 respectively.

## 5 Summary

By optimizing an exhaustive search algorithm, we find 13 new longest $k$-coils, 21 new longest $k$-snakes, and verify that some of them are optimal. By optimizing an algorithm by Hiltgen and Paterson [31] to find single-track circuit codes, we additionally find another 8 new longest single-track circuit codes and 8 new longest $k$-coils. Using these $k$-coils with some basic backtracking, we find 18 new longest $k$-snakes. Instances of each of these new longest $k$-coils and $k$-snakes are provided in the Appendix.

Future work in this area may be to apply the exhaustive search optimizations to both new and known heuristic approaches in the search for longer $k$-snakes and $k$-coils, especially when $k > 2$. Additionally, it would be interesting to study $k$-snakes and $k$-coils in the context of other graph classes.

## References


1. H. L. Abbott and M. Katchalski. On the snake in the box problem. *Journal of Combinatorial Theory. Series B*, 45(1):13–24, 1988.
2. H. L. Abbott and M. Katchalski. On the construction of snake in the box codes. *Utilitas Mathematica*, 40:97–116, 1991.
3. J. Bishop. Investigating the snake-in-the-box problem with neuroevolution. *Dept. of Computer Science, The University of Texas at Austin*, 2006.
4. B. P. Carlson and D. F. Hougen. Phenotype feedback genetic algorithm operators for heuristic encoding of snakes within hypercubes. In Martin Pelikan and Jrgen Branke, editors, *GECCO*, pages 791–798. ACM, 2010.
5. D. A. Casella and W. D. Potter. New lower bounds for the snake-in-the-box problem: Using evolutionary techniques to hunt for snakes. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, pages 264–269. AAAI Press, 2005.

6. D. A. Casella and W. D. Potter. Using evolutionary techniques to hunt for snakes and coils. In *IEEE Congress on Evolutionary Computing*, pages 2499–2505. Edinburgh, UK, 2005.
7. Y. Chebiryak and D. Kroening. An efficient SAT encoding of circuit codes. In *International Symposium on Information Theory and Its Applications*, 2008.
8. R. T. Chien, C. V. Freiman, and D. T. Tang. Error correction and circuits on the $n$-cube. In *2nd Allerton Conf. Circuit and System Theory*, pages 899–912, 1964.
9. D. Davies. Longest 'seperated' paths and loops in an $n$ cube. *IEEE Transactions on Eletronic Computers*, page 261, 1965.
10. K. Deimer. Some new bounds for the maximum length of circuit codes. *IEEE Transactions on Information Theory*, 30(5):754–756, 1984.
11. K. Deimer. A new upper bound on the length of snakes. *Combinatorica*, 5:109–120, 1985.
12. P. A. Diaz Gomez and D. F. Hougan. Genetic algorithms for hunting snakes in hypercubes: Fitness function analysis and open questions. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing 2006*, pages 389–394, 2006.
13. R. J. Douglas. Some results on the maximum length of circuits of spread $k$ in the $d$-cube. *Journal of Combinatorics Theory*, 6:323–339, 1969.
14. R. J. Douglas. Upper bounds on lengths of circuits of even spread in the $d$-cube. *Journal of Combinatorics Theory*, pages 206–214, 1969.
15. P. G. Emelyanov. On an upper bound for the length of a snake in an $n$-dimensional unit cube. *Diskret. Anal. Issled. Oper.*, 2(3):10–17, 1995.
16. P. G. Emelyanov and A. Lukito. On the maximal length of a snake in hypercubes of small dimension. *Discrete Mathematics*, 218(1–3):51–59, 2000.
17. T. Etzion and K. G. Paterson. Near-optimal single-track Gray codes. *IEEE Transactions on Information Theory*, 42:779–789, 1996.
18. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman San Francisco, 1979.
19. B. Harris. *The Use of Circuit Codes in Analog-to-Digital Conversion, Graph Theory and its Applications. Pages 121-132*. 1977.
20. A. P. Hiltgen and K. G. Paterson. Single-track circuit codes. *IEEE Transactions on Information Theory*, 47(6):2587–2595, 2001.
21. A. P. Hiltgen, K. G. Paterson, and M. Brandestini. Single-track Gray codes. *IEEE Transactions on Information Theory*, 42:1555–1561, 1996.
22. M. Juric, W Potter, and M. Plaskin. Using PVM for hunting snake-in-the-box codes. In *Proceedings of the Transputer Research and Applications Conference*, pages 97–102, 1994.
23. W. H. Kautz. Unit-distance error-checking codes. *IRE Trans Electronic Computers*, pages 179–180, 1958.
24. D. Kinny. A new approach to the snake-in-the-box problem. In Luc De Raedt, Christian Bessire, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas, editors, *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 462–467. IOS Press, 2012.
25. V. Klee. A method for constructing circuit codes. *Journal of the Association for Computing Machinery*, 14:520–528, 1967.
26. V. Klee. The use of circuit codes in analog-to-digital conversion. In *Graph Theory and its Applications*. B. Harris, Ed. New York: Academic, 1970.
27. V. Klee. What is the maximum length of a d-dimensional snake? *American Mathematics Monthly*, pages 63–65, 1970.
28. K. J. Kochut. Snake-in-the-box codes for dimension 7. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 20:175–185, 1996.
29. A. Lukito. An upper bound for the length of snake-in-the-box codes. In *6th International Workshop Algebraic and Combinatorial Coding Theory*, 1998.
30. A. Lukito and A. J. van Zantan. Stars and snake-in-the-box codes. Technical Report DUT-TWI-98-43, Dept. Tech. Math. Informatics, Delft Univ. Technol., Delft, The Netherlands, 1998.
31. K. G. Paterson and J. Tuliani. Some new circuit codes. *IEEE Transactions on Information Theory*, 44(3):1305–1309, 1998.
32. W. Potter. Latest records for the snake-in-the-box problem. In *http://www.ai.uga.edu/sib/records/*, 2012.
33. W. D. Potter, R. W. Robinson, J. A. Miller, K. Kochut, and D. Z. Redys. Using the genetic algorithm to find snake-in-the-box codes. In *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 421–426, 1994.
34. D. S. Rajan and A. M. Shende. Maximal and reversible snakes in hypercubes. In *24th Annual Australasian Conference on Combinatorial Mathematics and Combinatorial Computing*, 1999.
35. M. Schwartz and T. Etzion. The structure of single-track Gray codes. *IEEE Transactions on Information Theory*, 45:2383–2396, 1999.
36. R. C. Singleton. Generalized snake-in-the-box codes. *IEEE Transactions on Electronic Computers*, pages 596–602, 1966.
37. N. Sloane. The on-line encyclopedia of integer sequences. sequence number: A000937, A099155. In *http://oeis.org/*, 2012.
38. H. S. Snevily. The snake-in-the-box problem: A new upper bound. *Discrete Mathematics*, 133(3):307–314, 1994.
39. F. I. Solov'jeva. An upper bound for the length of a cycle in an $n$-dimensional unit cube. *Diskr. Analiz*, 45:71–76, 1987.
40. C. A. Taylor. A comprehensive framework for the snake-in-the-box problem. *Thesis for Master of Science, The University of Georgia*, 1998.
41. D. R. Tuohy, W. D. Potter, and D. A. Casella. A hybrid optimization method for discovering snake-in-the-box codes. In *First Symposium on Foundations of Computational Intelligence (FOCI'07)*, 2007.
42. D. R. Tuohy, W. D. Potter, and D. A. Casella. Searching for snake-in-the-box codes with evolved pruning methods. In *International Conference on Genetic and Evolutionary Methods*, pages 3–9, 2007.
43. J. Wojciechowski. A new lower bound for snake-in-the-box codes. *Combinatorica*, 9:91–99, 1989.
44. A. D. Wyner. Note on circuits and chains of spread $k$ in the $n$-cube. C-20(4):474–474, April 1971.
45. E. Wynn. Constructing circuit codes by permuting initial sequences. *CoRR*, abs/1201.1647, 2012.
46. Y. Yehezkeally and M. Schwartz. Snake-in-the-box codes for rank modulation. In *ISIT*, pages 2983–2987. IEEE, 2012.
47. G. Zémor. An upper bound on the size of the snake-in-the-box. *Combinatorica*, 17:287–298, 1997.
48. I. Zinovik, Y. Chebiryak, and D. Kroening. Periodic orbits and equilibria in glass models for gene regulatory network. *IEEE Transactions on Information Theory*, 56(2):1819–1823, 2009.
49. I. Zinovik, D. Kroening, and Y. Chebiryak. Computing binary combinatorial Gray codes via exhaustive search with SAT solvers. *IEEE Transactions on Information Theory*, 54(4):1819–1823, 2008.

# Appendices

### A Instances of longest $k$-coils and $k$-snakes

Each $k$-coil and $k$-snake can be represented compactly by its transition (or coordinate) sequence starting from some arbitrary starting vertex, generally assumed to be $0^n$. Assuming that the first bit position is 0 and that the symbols $a$ to $g$ represent the positions 10-16, we provide the transition sequences for each new longest $k$-coil and $k$-snake discovered. Each sequence is prefaced by $(n, k, t)$ where $n$ is the length of the bitstrings, $k$ is the spread, and $t$ is the length of the $k$-coil or $k$-snake. Observe that for $k$-snakes this means the transition sequence will have $t$ symbols, while for the $k$-coils the transition sequence will have $t-1$ symbols.

The $k$-coils (13,3,494), (14,3,812), (15,3,1380), (16,3,2240), (17,3,3910), (15,4,480), (16,4,768), and (17,4,1224) are also single-track circuit codes. However, to have the single-track property, the starting vertex is important. For these coils, we add in this starting string as a 4th parameter.

#### A.1 $k$-coils

**(9,3,64):** 012341536173215301738153612371530123415361732153017381536123715

**(10,3,102):** 01234153612371538123915301234153612391437153612341530123915624135018361234502371536123415301239143715

**(13,3,494,0000010010001):**1276b057b804a8905689c543cba90b13867450349802790a26cab278ab176510cb2035a8967256ba249b2c481c049ac0398732104257c
ab894780c46b0416a3126bc125ba9543264791c0ab69a216802638c5348013470cb7654869b312c08bc438a2485a1756a235692109876a8b053412a0165ac46a7c39
78c4578b432ba98ca0275634c2387c168c915b9a1679a06540cba1c2497856145a9138a1b370bc389bc2876210c3146b9a78367cb35ac30592015ab014a984321536
80bc9a5891057c1527b4237c0236cba6543758a201bc7ab327913749064591245810c876597ac423019c0549b3596b2867b3467a321a987b9c164523

**(14,3,812,00000010001111):**12831d0914d203b5248a9614031c0872a36254837a9284d739b037c13467d864506947582a79d10b6958635dc718b7a9d8c107d94c80
258c3689bc4db9a5b09cad71c04652b0adb8a43c6d2c104d365c4093d57ad38bd02394201a250314c6359ba725142d1983b47365948ba395084ac148d24578097561
7a58693b8a021c7a697460d829c8ba09d2180a5d91369d479acd50cab6c1adb082d15763c1b0c9b54d703d2150476d51a4068b049c0134a5312b361425d746acb836
25302a94c58476a59cb4a6195bd2590356891a86728b697a4c9b132d8b7a8571093ad9cb1a03291b60a247a058abd061dbc7d2b0c193026874d2c1dac65081403261
587062b5179c15ad1245b6423c4725360857bdc94736413ba5d69587b6adc5b72a6c036a14679a2b97839c7a8b5dac24309c8b96821a4b0adc2b143a2c71b358b169
bc01720cd803c1d2a413798503d20bd76192514372698173c628ad26b02356c7534d5836471968c0da5847524cb607a698c7b0d6c83b7d147b2578ab3ca894ad8b9c
60bd3541ad9ca7932b5c1b0d3c254b3d82c469c27ac

**(15,3,1380,000000000011011):**e3d5102316524cd3768ac081d5e489a7cba5c19bcd549d275c318c4a8201e46ab492876920ac23e75ca078c9ad8b04e6213427635de
4879bd192e6059ab8dcb6d2acde65ae386d429d5b9312057bc5a3987a31bd34086db189dabe9c1507324538746e0598ace2a30716abc9edc7e3bde076b0497e53ae6c
a423168cd6b4a98b42ce45197ec29aebc0ad2618435649857016a9bd03b41827bcda0ed804ce0187c15a8064b07db534279de7c5ba9c53d0562a80d3ab0cd1be37295
4675a968127bace14c52938cdeb10e915d01298d26b9175c18ec64538ae08d6cbad64e1673b91e4bc1de2c0483a65786ba79238cbd025d63a49de0c210a26e123a9e3
7ca286d290d75649b019e7dcbe7502784ca205cd2e03d1594b76897cb8a349dce136e74b5ae01d321b370234ba048db397e3a1e8675ac12a08edc08613895db316de3
014e26a5c879a8dc9b45aed0247085c6b012e432c481345cb159ec4a804b209786bd23b190ed197249a6ec427e04125037b6d98ab9edac56b0e1358196d7c1230543d
592456dc26a0d5b915c31a897ce34c2a10e2a835ab70d538015236148c7ea9bca0ebd67c1024692a7e8d2341654e6a3567ed37b1e6ca26d42b9a8d045d3b2103b946b
c81e649126347259d80bacdb10ce78d21357a3b809e345276507b46780e48c207db37e53cab9e156e4c3214ca57cd92075a23745836ae91cbdec21d089e32468b4c91
a0456387618c57891059d318ec48064dbca026705d4325db68dea3186b34856947b0a2dce0d32e19a043579c5da2b1567498729d689a216ae4290d59175ecdb13781
6e5436ec79e0b4297c45967a58c1b3ed01e4302ab15468ad6eb3c26785a983ae79ab327b053a1e6a2860dec248927065470d8a01c53a8d56a78b69d2c40e1205413bc
26579be70c4d37896ba94b08abc438c164b207b3971e0d359a38176581e9b12d64b9e67b89c7

**(16,3,2240,0000001001010001):**129347253468729f541f6a0d9802f867c8e17824a02c6012f41893769587ae4175a4f015973b02dc3169783defa168a2e547c6095eb
0f4dac3240826b97d036ac259d87e0698cd4ef2d0ef132d4a0fca15b843bda312739c23df5bd71bcdafc34e21403259fc205fabc042e6bd87ec1423e89a5c135d90f271b
4096baf857edfb3d16428be157d048329b14378f9ad8b9aced8f5ba75c063fe685ecd2e47de8a0682c6785a7ef9dcfbed04a7db0a567bfd91683297cfde934507ce084ba
d2c6fb4165a30298a6e8c1fd369c028bf3ed46cfe23a45836457983a065207b1ea9130978d9f28935b13d71230529a487a698bf5286b50126a84c13ed427a894ef0b27
9b3f658d71a6fc105ebd4351937ca8e147bd36ae98f17a9de5f03e1f0243e5b10db26c954ceb42384ad34e06ce82cdeb0d45f3251436a0d3160bcd153f7ce98fd2534f9a
b6d246ea10382c51a7cb0968fe0c4e27539cf268e15943ac254890abe9cabdfe906cb86d1740f796fde3f58ef9b1793d7896b8f0aed0cfe15b8ec1b678c0ea27943a8d0e
fa45618df195cbe3d70c5276b413a9b7f9d20e47ad139c04fe57d0f34b56947568a94b176318c2fba241a89ea039a46c24e8234163ab598b7a9c06397c61237b95d24fe
538b9a5f01c38ac40769e82b70d216fce5462a48db9f258ce47bfa9028baef6014f201354f6c21ec37da65dfc53495be45f17df93defc1e5604362547b1e4271cde26408
dfa90e36450abc7e357fb21493d62b8dc1a790f1d5f3864ad0379f26a54bd3659a1bcfadbce0fa17dc97e285108a70ef4069f0ac28a4e89a7c901bfe1d0f26c9fd2c789d1
fb38a54b9e1f0b56729e02a6dcf4e81d6387c524bac80ae31f58be24ad150f68e1045c67a58679ba5c287429d30cb352b9afb14ab57d35f9345274bc6a9c8bad174a8d7
2348ca6e350f649cab6012d49bd5187af93c81e3270df6573b59eca0369df58c0ba139cbf07125031246507d32fd48eb76e0d645a6cf56028e0a4ef0d2f6715473658c2
f5382def37519e0ba1f47561bcd8f4680c325a4e73c9ed2b8a102e604975be148a037b65ce476ab2cd0becdf10b28eda8f396219b81f0517a01bd39b5f9ab8da12c0f2e1
037da0e3d89ae20c49b65caf201c6783af13b7ed05f92e7498d635cbd91bf42069cf35be261079f2156d78b6978acb6d39853ae41dc463cab0c25bc68e460a456385cd7
bad9cbe285b9e83459db7f461075adbc7123e5ace6298b0a4d92f4381e07684c6afdb147ae069d1cb24adc018236142357618e430e59fc87f1e756b7d067139f1b5f01e
307826584769d306493ef04862af1cb2058672cde905791d436b5f84dafe3c9b213f715a86cf259b148c76df587bc3de1cfde021c39feb904a732ac9201628b12ce4ac60
abc9eb23d103f2148eb1f4e9abf31d5ac76db0312d7894b024c8fe160a3f85a9e746dcea2c05317ad046cf37218a03267e89c7a89bdc7e4a964bf52ed574dbc1d36cd79f
571b567496de8cbeadcf396caf9456aec80572186becd8234f6bdf73a9c1b5ea305492f18795d7b0ec258bf17ae2dc35be

**(17,3,3910,00000101000101001):**12b40756189g7f41258d1eg032fg01d30c6g5732ac8b16adb97afd8910de79cfabd4e9bd32610c751bea3921dce9bg19ae246ag734
968ab280a65d3gc10gf43agd4901e7d4efg47059abf93b0f8cde6f8e2b673bf89054gf78g2394e7248gb0346e9bd6fb17f516e054g61b8f4156439eac19034d629783ab
1906347af3g12540123f52e817954cead38cfdb9c0fab32fg9be0cdf6gbdf54832e973dgc5b43fegbd13bcg468c1956b8acd4a2c87f51e321065c1f6b23g9f6g016927bc
d0b5d20aefg80ag4d895d0ab276109a145b6g946a1d2568gbdf80d390738g276183da0637865bgce3b256f84b9a5cd3b28569c05134762345074ga39b76egcf5ae0fd
be20cd5401bdg2ef081df076a54gb95f1e7d650g1df35de168ae3b78dacef6c4ea9073g543287e308d451b081238b49def2d7f42cg01a2c16fab7f2cd49832bc367d81b
68c3f478a1df0a2f5b295a14983a5fc2859a87d1eg5d4780a6dbc7ef5d4a78be273569845672961c5bd98g1e07cg20fdg42ef7623df14g02a3f0298c761db703g9f8721
3f057fg38acg5d9afceg08e6gcb29517654a9g52af673d2a345ad6bfg04f9064e123c4e380cd904ef6ba54de589fa3d8ae5069ac3f02c407d4b7c36ba5c70e4a7bca9f3g1
7f69a2c8fde9g07f6c9adg49578ba67894b83e7dfba13g29e1420f164g09845f036124c5024bae983fd9251b0a94350279015ace17fbc0eg12ag81ed4b739876cb174c0
895f4c567cf8d01260b286g345e6g5a2efb26g08dc76fg7ab0c5facg728bce5024e629f6d9e58dc7e92g6c9decb0513908bc4ea0fgb12908ebcf16b79adc89ab6da5g9f0
dc3514bg3642038612ba670258346e7246dcgba50fb473d2cb657249b237ceg390de2g134c1a3gf6d95ba98ed396e2ab706e789e0af23482d4a81567g817c4g0d4812
afe98019cd2e70ce194adeg7246g84b08fbg7afe9gb418ebfged2735b2ade6gc201d34b2agde038d9bcfeabcd8fc71b02fe5736d1586425a834dc89247a568g9468fe1dc
720d695f4ed87946bd459eg15b2fg41356e3c5108fb7dcbagf5b8g4cd928g9abg2c0456a4f6ca37891a39e612f6a34c0gba23bef4g92eg3b6cfg194681a6d2a0d19c0gb
1d63agd01gf4957d4cfg81e423f56d4c1fg25afbde0gcdefa0e93d240g7958f37a8647ca56feab469c78a1b68a0g3fe942f8b706gfa9b68df67bg137d40163578g5e732a
0d9fedc107da16efb4a1bcd14e2678c608ec59ab3c5bg83408c56e21dc45dg061b4g15d8e013b68a3c8f4c2f3be21d3f85c1f23106b79f6e01a3g645078f6e30147c0dfg
21efg0c2gb5f46219b7a059ca869ec780gcd68be9ac3d8ac2150gb640ad92810cbd8af089d1359f6238579a17g954c2fb0gfe329fc38g0d6c3def36g489ae82age7bcd5e

7d1a562ae78g43fe67f1283d6137fag235d8ac5ea06e405dg43f50a7e3045328d9b08g23c5186729a08g52369e2f0143g012e41d706843bd9c27beca8bge9a21ef8adgb
ce5face43721d862cfb4a32edfac02abf357b0845a79bc391b76e40d210g54b0e5a12f8e5fg05816abcga4c1g9def7g9f3c784cg9a1650g89034a5f83590c1457face7gc2
8g627f165072c9g526754afbd2a145e73a894bc2a17458bg4023651234g63f928a65dfbe49dgecad1gbc43g0acf1deg70ceg65943fa84e0d6c54gf0ce24cd0579d2a67c
9bde5b3d98g62f432176d2g7c340ag70127a38cde1c6e31bfg091b05e9a6e1bc38721ab256c70a57b2e36790ceg91e4a18490387294eb1748976c0df4c367g95cab6de
4c3967ad162458734561850b4ac87f0dg6bf1gecf31de6512ce03fg192eg187b650ca6g2f8e76102eg46ef279bf4c89ebdfg7d5fba1840654398f419e562c192349c5aef
g3e8g53d012b3d27gbc8g3de5a943cd478e92c79d4g589b2eg1b3g6c3a6b25a94b6gd396ab98e2f06e5891b7ecd8fg6e5b89cf38467a956783a72d6cea902f18d031ge
053fg8734eg25013b4g13a9d872ec8140ag98324g168g049bd06eabgdf019f70dc3a628765ba063bg784e3b456be7cg015ga175f234d5f491dea15fg7cb65ef69agb4e
9bf617abd4g13d518e5c8d47cb6d81f5b8cdbag4028g7ab3d9gefa018g7dabe05a689cb789a5c94f8egcb2403af2531g27501a956g147235d6135cbfa94gea362c1ba54
6138a126bdf28gcd1f023b092fe5c84a987dc285d19a6g5d678dg9e12371c3970456f706b3fgc37019ed87g08bc1d6gbd0839cdf6135f73ag7eaf69ed8fa307daefdc162
4a19cd5fb1g0c23a19fcdg27c8abed9abc7eb60ag1ed4625c04753149723cb781369457f8357ed0cb61gc584e3dc76835ac348df04a1ef30245d2b40g7ea6cba9fe4a7f3
bc817f89af1bg34593e5b926780928d501e5923bgfa912ade3f81df2a5bef08357095c19gc08bgfa0c529fcg0fe3846c3bef70d312e45c3b0ef149eacdgfbcde9gd82c13g
f6847e2697536b945ed9a358b6790a579gf2ed831e7a6g5fe98a57ce56af026c3g052467f4d6219gc8edcb0g6c905dea390abc03d1567b5g7db489a2b4af723g7b45d1
0cb34cfg50a3f04c7dg02a5792b7e3b1e2ad10c2e74b0e120g5a68e5dg092f534g67e5d2g036bgcef10defgb1fa4e35108a69g48b9758db67gfbc57ad89b2c79b104gfa
53g9c8170gbac79eg78c0248e5127468906f843b1eagfed218eb27fgc5b2cde25f3789d719fd6abc4d6c094519d67f32ed56e0172c5026e9f124c79b4d9g5d3g4cf32e4
g96d2g34217c8ag7

**(11,4,70):** 01234561735869152761 0a28604726314560748529376 25a98231780a9246570a4873

**(12,4,102):** 01234561375806352091650 71a63012650417608126509 2317b0247168249a1736259712067329876124a6531a76925a614ba

**(15,4,480,000010001101101):**19a7b2e306ad748a91c30a21d5cea2dbe78590c1d48b52687ea1d80eb3ac80b9c5637daeb26930465c8eb6dc918a6d97a3415b8c9047
1d243a6c94ba7e684b75812e396a7d25eb02184a72985c4629536e0c17485b03c9d0e62850763a2407314cdae52639d1a7bdc4063d541802d51e2ab8c30417be859b
a2d41b32e6d0b3ec0896a1d2e59c637980b2e910c4bd91cad6748eb0c37a41576d90c7eda29b7ea8b4526c9da1582e354b7da5cb80795c8692304a7b8e360c13295b8
3a96d573a64701d28596c14dae1073961874b3518425deb06374ae2b8ced5174e652913e6203bc9d41528c096acb3e52c4307e1c40

**(16,4,768,0000010001001101):**14a90f23789cf6d3406c9e4dc53ad0631dbce0148f5b1d9ae176dcf04569c3a01d396b1a9207ad30ea89bde15c28ea67be43a9cd1236
907dea0638e76fd47a0db7568abe29f5b7348b10769aef036d4ab7d305b43ca147da8423578bf6c2840158ed4367bcd03a1784a0d281097e14a751f02458c39f51de25
ba103489ad07e4517daf5ed64be1742ecdf125906c2eabf287ed01567ad4b12e4a7c2ba318be41fb9acef26d39fb78cf54bade2347a18efb1749f870e58b1ec8679bcf3a0
6c8459c2187abf0147e5bc8e416c54db258eb9534689c07d3951269fe5478cde14b2895b1e3921a8f25b862013569d4a062ef36cb21459abe18f5628eb06fe75cf2853f
de0236a17d3fbc0398fe12678be5c23f5b8d3cb429cf520cabdf037e4a0c89d065cbef3458b29f0c285a0981f69c2fd978acd04b17d956ad3298bc01258f6cd9f527d65e
c369fca64579ad18e4a6237a0f6589def25c39a6c2f4a32b9036c97312467ae5b173f047dc3256abcf2906739fc170f86d039640ef1347b28e40c

**(17,4,1224,00000010010100011):**2134a80b5a31206b74g0582b10f2e5879gb10c4f256gc104f3701b865cd80b1935g0e8d27689fd5gaf8675bgc945ad7g65372adce4
g650937ab4065938c56gdba01d5g6e8a452d17cbde31a4f3dbcag40e9af1c4ba8c7f10294ba5e8cfg95bae8d0ab41gf561a4b2df9a716c0g1286f9381g0f4952ef3609gf
d0c3657e9gfa2d034eagf2d15fg9643ab6f9g713efc6b05467db3e8d64539ea7238b5e431508bac2e43f715892f43716a34eb98fgb3e4c68230bg5a9bc1g82d1b9a8e2fc
78dga2986a5dgf072983c6ade7398c6bf892ged34g8290bd785g4afeg064d716gefd2730cd14f7edbfa1435c7ed80bf12c8ed0bg3de7421894d7e5g1cda49f3245b91c6
b42317c8501693c21g3f698a0c21d5g3670d215g4812c976de91c2a4601f9e3879age60bg9786c0da56be8076483bedf50761a48bc5176a49d670ecb12e607f9b563e2
8dcef42b5g4ecdb051fabg2d5cb9d8g213a5cb6f9dg0a6cbf9e1bc520g672b5c3egab827d102397ga49201g5a63fg471a0ge1d4768fa0gb3e145fb0g3e26g0a754bc7ga
0824fgd7c16578ec4f9e7564afb8349c6f542619cbd3f54g8269a3g54827b45fca9g0c4f5d79341c06bacd2093e2cab9f3gd89e0b3a97b6e0g183a94d7bef84a9d7cg9a3
0fe45093a1ce89605bgf0175e8270fge3841de25g8fecgb2546d8fe91cg23d9fe1c04ef85329a5e8f602deb5ag4356ca2d7c53428d96127a4d3204g7a9b1d32e604781e
32605923da87efa2d3b5712gaf498ab0f71c0a897d1eb67cf9187594cfeg61872b59cd6287b5ae781fdc23f718gac674f39edfg53c605fdec162gbc03e6dcae90324b6dc
7gae01b7dcgaf2cd6310783c6d4f0bc938

**(12,5,60):** 012345678239a562b9145678910a7631204567820ba5629b145678b13a7

**(13,5,80):** 012345671824951a640253a4b0c382401932a709b2640719cb3754192a38c12097a14230b762a3c

**(14,5,106):** 0123456718395012345 60a2b35c012345608319507329ad7062ca73125a43085942635104c36a0b423506143870c1359721ca569d

**(14,6,68):** 01234567829a45b702c4adb301ca9b7601892570cd12473ad82970ab68475d3a64c

**(15,6,88):** 012345607182930a142b35061743c085d123046915703e9ca10426bc134508a9c765421cabd76350417c8b5

**(16,6,118):** 012345607182930a142b35061743c085d1230469157038a1420e361f25073b12408691370d582a74019e3876dc210e67b319a246735d0a19fb75a

**(16,7,76):** 0123456708192a3b041c253d06172e480597bf18c4a92307c86e24b958012cea4870d62afb7

**(17,7,102):** 0123456708192a3b041c253d06172e48031f259406172a3b051g789dc0b324ed109ab76de2c0153be89a0124cefd3b072e95d

**(15,7,60):** 2e571b9ac682531946cd5e17402cb6184d9c2e1a5d8327eb1c682470bd9

## A.2 $k$-snakes

**(8,3,35):** 01230425614273614067153762541650341

**(9,3,63):** 012304250123062704135423041356280524032105240726531403245314072

**(10,3,103):** 1536123415301239156241350183612345023715361234153012391437153012341536123715381239153012341536123915302

**(11,3,157):** 0123415630782098 67a527958315037862180239425983714351 90a1203984a924056415903a601a527a42059675462186a1520782a71437642158316
84a987a14239473a603864a190a89675937a

**(12,3,286):** 0123456708297109478309120879 01465a7b685b06a95b875a6b580413624a12543b12a613421a508947038710928734890 78315ab0659a685b7a690a
b56a98132541b34a12634b532143ba897108290387490219780923ab685a7b59a60b578b6a5b79324a13621b345216a243126b970389478290178437098742b659a
b06a7b586a0965ba607241b32543621a43

**(13,3,493):** 0123456705189ab147892460c4a9c579417c640c7814ca54b2063b1a2071c5916094a01ba9c159a2b19641375c83b675ab940bc50165b3609b40673b0c
1b8a49283ca4630153945bc438c50315ca8359b32610728961c85b48014391829458b4962840387cb5a720cb9243125b180b270142310c721582a9346a75930718b
743b2537a5b178b59a7b4276081c6a4085ab23a183748a643ba23406a317ac52b9c6152463786b28a126c18367815c68ba6947309cb471c8a2c3726b7c9b28ca2b49
c236c01a850931ab926798a7c3a90372967310978c95b624508b6307ca026a986058a70ca8b50a29043c714523c85a9657c602c5426a

**(14,3,812):** 0123456789651270 61a82b9164cd568ab083951a640d719084c298c160c8ba327901725b9d7580bc1573462708cd0a125873d6951d034b10457d40c8a
b91d59b2c16920dc4529a37b9d046d85b209a671256da3c5d32963d408c15621cb4571bd6432b18a9c16d37602cbd18795b2768a426ab17a63d04527b54c3295c673
abc50814576a97db4c650912cb97083b78c5987a6d32b9c234ab12479a8c42d0532978196c3472d15b4c19d0ac9042109876abc14ba38c5b3918043b6d2ab1905174
a39b652c34516d841d3b5d10978c453c8a042ca150d3ac76b8c51d25938a1c72b4a325760356ac265d190432a408d3b4852d6a8497c04256b21a08549bc38ab297da
2784b72651d3ab83d06ac302b67803194d3b27cb58d0231c4a08cb1968b903c9b7256a8c0a6d784adbc790da5136acb94c206dba5438d04c5170c1da41c9b27804d8
7690386c491d6825a784c134bd76c823a06d34259d45683541cb90d360971da074315670b2890345a3c69740ba8d76a3b2163270a2354c1d6a7d19568d93a5279dcb
01da328a47193dc806978acb57ab9d

**(15,3,1380):** 012345674891a48bc69b2d74a6b5ad35014e8b0e5c8d506c5168ab20eda756a913e5b409d347d0e326158ac1928cb9e4dc82da091da36803e72d183c17
4cb9d5a3b5c6a7cb86c98a13eb5712c810945c3db074d27b54e89ca1690ea6305d76ae71b097148ab452e79a4692d6307c143c681263a860a19453c29e6a9b0dc6473
b2d7e23cd5a061980b5184bc72815293b029da13dce5201d80e784b269d468a9e841a8b190dc46e058103b768d243e725e467c1b890ab3c9ad362ea9ce043be071947
65ceb97ab52ad3e807d8a105ad91a390b76d85bca9b4328a7ed452ec5d82693a0b134601748e51065bd435b290d28c6530213ce1745ab27a19bc1709140b3287ac361

03d4ea1257dce56c7ae8041b394d8b92da5c9b8c37d4c3e0b7ea68c4be946592dc13e21903692b09db34ea216489b47d519ec2765c86215abd9340d7a30e71c603a64
27d645b32518a6d350d8c0e76945e90b480e3b0734d51e98da03d27c9056e28c6a8e9c13704db7214b52968b418de278dc34ec9a1874cb7a6b5280dc50b3dab543b2
4d7c950a71b47e260bc85ea681a506942bd732e9d3ce08a3d9a75e2a764d56019a2d632183ceab76cb34713cd43ed7260cb1293d25e8b36ac518a91cb80de3724e507
465ba147012c5e128d7c8b901e784e9a4651328634d29467d4572e8b639e047ec5a34816c9a10963ab7542ed5cb2d8c319d2b9e6c59ea726a30b952ad501d8c94ea84
d7e0d827dc2e5a38405bd256c14da986019b084132cde57c63e7a64907e30586c0512e814b30ce17cb97a60d51ad725b7ae276e5c14adbc37ec869d710a8b903bad94
e675c26845218d0b254bca86bc9e5a9d34b6592630218b7c9172ec3215e285c69d17364256a80729b1a30b43170d582c6e8adce9a7b3ecd361a83605c1074d38c0e84
be9a326092e564e9c5eac68079248dec81ab2e03914b3d492b7cae685a194

**(16,3,2240):** 1f6a0d9802f867c8e17824a02c6012f41893769587ae4175a4f015973b02dc3169783defa168fc3e497c02a136de27df4107af49adeb0ad4b2c3f528baef6
014f201354f6c21ec37da65dfc53495be45f17df93defc1e5604362547b1e4271cde26408dfa90e36450abc7e35c90b1649df7e03abf4ac1ed47c167ab8d7a18f90c2f587
bc3de1cfde021c39feb904a732ac9201628b12ce4ac60abc9eb23d103f2148eb1f4e9abf31d5ac76db0312d7894b0296d8e316ac4bd078c179eba149e34785a47e5c6d9
fc254890abe9cabdfe906cb86d1740f796fde3f58ef9b1793d7896b8f0aed0cfe15b8ec1b678c0ea27943a8d0efa45618df63a5b0e37918ad459e46b87e16b01452714b2
93a6c9f2156d78b6978acb6d39853ae41dc463cab0c25bc68e460a456385cd7bad9cbe285b9e83459db7f461075adbc7123e5ac30728db046e57a126b13854be38de12
f4e18f607396cfe23a45836457983a065207b1ea9130978d9f28935b13d71230529a487a698bf5286b50126a84c13ed427a894ef0b2790d4f5a8d13b247ef38e05218b
05abefc1be5c3d40639cbf07125031246507d32fd48eb76e0d645a6cf56028e0a4ef0d2f6715473658c2f5382def37519e0ba1f47561bcd8f46da1c275ae08f14bc05bd2
fe58d278bc9e8b290a1d30698cd4ef2d0ef132d4a0fca15b843bda312739c23df5bd71bcdafc34e21403259fc205fabc042e6bd87ec1423e89a5c13a7e9f427bd5ce189d
28afcb25af45896b58f6d7ea0d3659a1bcfadbce0fa17dc97e285108a70ef4069f0ac28a4e89a7c901bfe1d0f26c9fd2c789d1fb38a54b9e1f0b56729e074b6c1f48a29be5
6af57c98f27c12563825c3a4b7da03267e89c7a89bdc7e4a964bf52ed574dbc1d36cd79f571b567496de8cbeadcf396caf9456aec80572186becd8234f6bd41839ec157f
68b237c24965cf49ef2305f2907184a7d0f34b56947568a94b176318c2fba241a89ea039a46c24e8234163ab598b7a9c06397c61237b95d24fe538b9a5f01c38a1e506b
9e24c358f049f16329c16bcf0d2cf6d4e5174adc018236142357618e430e59fc87f1e756b7d067139f1b5f01e307826584769d306493ef04862af1cb2058672cde9057e
b2d386bf19025cd16ce30f69e389cdaf9c3a1b2e417a9de5f03e1f0243e5b10db26c954ceb42384ad34e06ce82cdeb0d45f3251436a0d3160bcd153f7ce98fd2534f9ab6
d24b8fa0538ce6df29ae39b0dc36b0569a7c6907e8fb1e476ab2cd0becdf10b28eda8f396219b81f0517a01bd39b5f9ab8da12c0f2e1037da0e3d89ae20c49b65caf201c
6783af185c7d2059b3acf67b068da9038d23674936d4b5c8eb14378f9ad8b9aced8f5ba75c063fe685ecd2e47de8a0682c6785a7ef9dcfbed04a7db0a567bfd91683297c
fde934507ce5294afd268079c348d35a76d05af0341603a18295b8e1045c67a58679ba5c287429d30cb352b9afb14ab57d35f9345274bc6a9c8bad174a8d72348ca6e3
50f649cab6012d49b2f617caf35d469015a02743ad27cd01e3d07e5f6285bed129347253468729d8b

**(17,3,3941):** e79cfabd4e9bd32610c751bea3921dce9bg19ae246ag734968ab280a65d3gc10gf43agd4901e7d4efg47059abf93b0f8cde6f8e2b673bf89054gf78g2394
e7248gb0346e9bd6fb17f516e054g61b8f4156439eac19034d629783ab1906347af3g12540123f52e817954cead38cfdb9c0fab32fg9be0cdf6gbdf54832e973dgc5b43
fegbd13bcg468c1956b8acd4a2c87f51e321065c1f6b23g9f6g016927bcd0b5d20aefg80ag4d895d0ab276109a145b6g946a1d2568gbdf80d390738g276183da063786
5bgce3b256f84b9a5cd3b28569c05134762345074ga39b76egcf5ae0fdbe20cd5401bdg2ef081df076a54gb95f1e7d650g1df35de168ae3b78dacef6c4ea9073g543287
e308d451b081238b49def2d7f42cg01a2c16fab7f2cd49832bc367d81b68c3f478a1df0a2f5b295a14983a5fc2859a87d1eg5d4780a6dbc7ef5d4a78be2735698456729
61c5bd98g1e07cg20fdg42ef7623df14g02a3f0298c761db703g9f87213f057fg38acg5d9afceg08e6gcb29517654a9g52af673d2a345ad6bfg04f9064e123c4e380cd90
4ef6ba54de589fa3d8ae5069ac3f02c407d4b7c36ba5c70e4a7bca9f3g17f69a2c8fde9g07f6c9adg49578ba67894b83e7dfba13g29e1420f164g09845f036124c5024ba
e983fd9251b0a94350279015ace17fbc0eg12ag81ed4b739876cb174c0895f4c567cf8d01260b286g345e6g5a2efb26g08dc76fg7ab0c5facg728bce5024e629f6d9e58
dc7e92g6c9decb0513908bc4ea0fgb12908ebcf16b79adc89ab6da5g9f0dc3514bg3642038612ba670258346e7246dcgba50fb473d2cb657249b237ceg390de2g134c1
a3gf6d95ba98ed396e2ab706e789e0af23482d4a81567g817c4g0d4812afe98019cd2e70ce194adeg7246g84b08fbg7afe9gb418ebfged2735b2ade6gc201d34b2agde0
38d9bcfeabcd8fc71b02fe5736d1586425a834dc89247a568g9468fe1dc720d695f4ed87946bd459eg15b2fg41356e3c5108fb7dcbagf5b8g4cd928g9abg2c0456a4f6c
a37891a39e612f6a34c0gba23bef4g92eg3b6cfg194681a6d2a0d19c0gb1d63agd01gf4957d4cfg81e423f56d4c1fg25afbde0gcdefa0e93d240g7958f37a8647ca56feab
469c78a1b68a0g3fe942f8b706gfa9b68df67bg137d40163578g5e732a0d9fedc107da16efb4a1bcd14e2678c608ec59ab3c5bg83408c56e21dc45dg061b4g15d8e013
b68a3c8f4c2f3be21d3f85c1f23106b79f6e01a3g645078f6e30147c0dfg21efg0c2gb5f46219b7a059ca869ec780gcd68be9ac3d8ac2150gb640ad92810cbd8af089d13
59f6238579a17g954c2fb0gfe329fc38g0d6c3def36g489ae82age7bcd5e7d1a562ae78g43fe67f1283d6137fag235d8ac5ea06e405dg43f50a7e3045328d9b08g23c518
6729a08g52369e2f0143g012e41d706843bd9c27beca8bge9a21ef8adgbce5face9bc391b76e40d210g54b0e5a12f8e5fg05816abcga4c1g9def7g9f3c784cg9a1650g89
034a5f83590c1457face7gc28g627f165072c9g526754afbd2a145e73a894bc2a17458bg4023651234g63f928a65dfbe49dgecad1gbc43g0acf1deg70ceg65943fa84e0d
6c54gf0ce24cd0579d2a67c9bde5b3d98g62f432176d2g7c340ag70127a38cde1c6e31bfg091b05e9a6e1bc38721ab256c70a57b2e36790ceg91e4a18490387294eb17
48976c0df4c367g95cab6de4c3967ad162458734561850b4ac87f0dg6bf1gecf31de6512ce03fg192eg187b650ca6g2f8e76102eg46ef279bf4c89ebdfg7d5fba1840654
398f419e562c192349c5aefg3e8g53d012b3d27gbc8g3de5a943cd478e92c79d4g589b2eg1b3g6c3a6b25a94b6gd396ab98e2f06e5891b7ecd8fg6e5b89cf38467a956
783a72d6cea902f18d031ge053fg8734eg25013b4g13a9d872ec8140ag98324g168g049bd06eabgdf019f70dc3a628765ba063bg784e3b456be7cg015ga175f234d5f4
91dea15fg7cb65ef69agb4e9bf617abd4g13d518e5c8d47cb6d81f5b8cdbag4028g7ab3d9gefa018g7dabe05a689cb789a5c94f8egcb2403af2531g27501a956g147235
d6135cbfa94gea362c1ba546138a126bdf28gcd1f023b092fe5c84a987dc285d19a6g5d678dg9e12371c3970456f706b3fgc37019ed87g08bc1d6gbd0839cdf6135f73a
g7eaf69ed8fa307daefdc1624a19cd5fb1g0c23a19fcdg27c8abed9abc7eb60ag1ed4625c04753149723cb781369457f8357ed0cb61gc584e3dc76835ac348df04a1ef30
245d2b40g7ea6cba9fe4a7f3bc817f89af1bg34593e5b926780928d501e5923bgfa912ade3f81df2a5bef08357095c19gc08bgfa0c529fcg0fe3846c3bef70d312e45c3b0
ef149eacdgfbcde9gd82c13gf6847e2697536b945ed9a358b6790a579gf2ed831e7a6g5fe98a57ce56af026c3g052467f4d6219gc8edcb0g6c905dea390abc03d1567b5
g7db489a2b4af723g7b45d10cb34cfg50a3f04c7dg02a5792b7e3b1e2ad10c2e74b0e120g5a68e5dg092f534g67e5d2g036bgcef10defgb1fa4e35108a69g48b9758db6
7gfbc57ad89b2c79b104gfa53g9c8170gbac79eg78c0248e5127468906f843b1eagfed218eb27fgc5b2cde25f3789d719fd6abc4d6c094519d67f32ed56e0172c5026e9
f124c79b4d9g5d3g4cf32e4g96d2g34217c8ag7f12b40756189g7f41258d1eg032fg01d30c6g5732ac8b16adb97afd89fe06gbacg2b1c02f1d0ef7d5e974539841

**(9,4,28):** 0123405167201487034572016540

**(10,4,47):** 01234056374180362405679241763257402187562135471

**(11,4,68):** 01234051620713280412905137021674380a54761045218902374801269548079410

**(12,4,104):** 01234051620713280412905137021674380a547610452189b2315029710824a01623048720194320ab174503b2408623017964a0

**(13,4,181):** 0123456738596ab0879c517ac3546ba132c01427c95640782b098123c59b1a86b3a0872c3604a5674ba18275b94c519640a81c6392c0359b4a0257382b
7c3694b8c17a861275396a2014a5081c73548b094cba0217c9a6b3927a3

**(14,4,279):** 01234567189a27680a7143b58c13bd64531da43b807cd9b07218c0b24807da39267a35bd9a758da3240651340c72643cd24058a1cb08a9351809258acd
4b97ad460cbda65cd49287634281a97241c92865d31085db46358b965d1c20badc27810cd761c2b95a742953dba9231b9576c43856c0274650b76c31980dc19a5381
ca73190b6da29b64c0db9430b

**(15,4,481):** 012345167894ab790184390a57c24ad7e98d04b9a7105da1cb7061c54a3805ea217ec0d15abc4e5b3dac9b340567c4258ba23ceb45d3024d6e531d60c49
a30847d58632d04e6c80e9246be9c30156c70ae47968ec02937c21809d2136cb493ac520a19723c816a38b7c1e8b693d0165348c5b1a8637b9567da3b27d916ecb946
0734db5796ad149ae56d8ae1b923d109ca60ed4a195eb015249e752bd186ebc1359c2e05b142dcb48012a48deb792d3b641382c4db08e3d07cb8507e2da18e6d90b67
830edc726eca3d74ca28e5b729e1cd9a76c2e3a892356ea0358724da812b3e15a93826571864925c647a80e57b8

**(16,4,767):** 01234567849a6bcd3e6a19ed7810ef4c2a85c4f75ed3b94753028fd6c841dabe29d1349e6c35978b01cfb876f9e2a486f250c7edbc83e1a904e32849db2f4
6ca53b7ac6d749018cd70f5b69eabc2931458920c84ea078db1f2a61bde68453cbe657fad491ab04238fc405bc89156cea3701d3ae9dc8f2ba9df671e8431a5802c7b85
fabc43fdb912653e2194ebc70a14e7d639c8231fc50b6acf71ab827ea430df2903489ab6513896ed24bc0237bf5ad1b7631ac0691825e704528c41adf32c4d9e08ab502
6a7f1e3a6d231b5d43c0f9658f0cb831e720b8e495c1af50d1673921de023afe82b574dfc75bac239605ac984fb317f5e3d62403e9502179c0af68e7b6fa1b024d5f1b4c
87a2367f92ed085294f50364b517dc96ad713a508ef73a8bc6102d67409e5cf0487f52d8af36eb4d1e6321f5c97621cabd350ed68549fb758c67f0ec172d9a8e39d2037
fb46d03b1ae2f59edcf847a6fcbd6759b360e41c924e05267a8de52a31907f49eb7c861d7baed6f4a2d5983b40895f0d6159

**(17,4,1224):** 012345621789a23167b5104c9ad8ec3209712fb06e3c278f1d4873614a258da023f17d32a5149b7a8g3e1b48agd78c95e103254bceg8b6dca245bg3680
93g4a891b7301cb468g04b1789edg13f428d931f0g35e728c4b79d52f3da051b97df4a3ce4f913e8dg4c85d9a3fc9d8g3e20f8469b30e486cf472gb359dge07b6401c78
deg069145296e84230f95370e1465e03f42bc639aed4c293a569gbfd47e0f2cgda9c85g302fcae897bea239b4c6e74gc289a72c469bd5a4e12095be417aefd609g2c6b5
f01e537f4cb65123egd21b4ed95a2g9f5b3e1gb59aed071928bce7d298g1260acefb5ad76c8274g695da78b42f0b8d920e71bfe67d428fd7e120cg8eb3d52g0be3f8bac
1526d710ga53bg9fae701g3d9b6cd30ebc2g8d62ag09b360g28bc5f32d407bfcd2463d1587ba0g8cf174dfe612gc8f40eda504c2d5bf30ab1fced4acfb3d5764b09cgd6
50b9a40873gd1cf3568g9062a8bf5369c2017c95b07d64c1d865209156d407ga9dce5f0a7cde19c3g4f085647a3fecab13d674ae5bc8g5e7dcg0a95803a7bce87a09cgf
1e05276c1g5028e54f96c37a9g146251d840ag9127d53f72g05fc1e73c41gd523g1ce5f682c7bga58f7cb32796ea54g1ef89ab78039c1fe8bg0746gbfc76582g4598f07
b4f85276a3b5gdf1736g5d4bgea2179f8263e1dg3c4e58623dfcg9afd65ga73bf97e36cgd9637bga14d7f068g4af709df21b8ge63ba4280f459273ab4065fe160a7f1g4d

6eg24a5f0ea4gdf1890g6ca3f916gce06b8d3f2a4d19b3c697ebg41d9ca7628ac1g68f90a2fb9176c219f0683ecfa5146e8af52cad3046b1908ed45aeg2df980e51gab3158fa36ec1b6de8ga5b8

**(10,5,25):** 0123450627385921705432760

**(11,5,39):** 012345671285970a814576a83425a9641570963

**(12,5,56):** 012345671285970a81479b5867a9218754361897a0145b9761428b71

**(13,5,79):** 0123450617285091325406la2b03172c045b239654013894ba136709435a6c29173ca897064a8b5

**(14,5,112):** 0123450617385091234056la2073b124053812c063192704135780a1273d069581c63052469850c73695b814ac98523c4795086391428596

**(15,5,206):** 7bde5a0b792e3b0467d38402a7b48ce06b1c8a304c1578e49513b8c59d017c2d9b415d268905a624c9d6ae128d3eac526e379a16b735dae7b0239e40bd637048ab27c846eb08c134a051ce748159bc38d9570c19d245b162d085926acd49ea681d2ae356c273e4

**(16,5,285):** 94f7cd02e8a36c05b72d80569b713cf59e40b619ef240ac58e27d94fa278bd935e17b062d83b01462ce7a049fb61c49adfb57039d284fa5d23684e09c26b1d83e6bcf1d7925bf4a61c7f458a602be48d3fa508de13f9b47d16c83e9167ac824d06af51c72af0351bd69f38ea50b389cea46f28c173e94c12573df8b15a0c72d5abe0c6814ae3950b6e34795f1ad48

**(17,5,473):** 34fgac96e287c19fb547g23fdc7g8934f0eb27dc06e3ga9c478310c4de89352g7c105b284fe09cd8650912de8a74c065ag7d9325e01dba5e6712dfc905baf4c1e87a2561gfa2bc67130e5agf39062dcf7ab643f7g0bc6852af438e5b7103cfgb983c45g0bda7f398d2agc658034ged809a45g1fc38ed17f40bad589421d5ef9a46308d216c395gf1ade9761a23ef9b85d176b08ea436f12ecb6f7823egda16cbg5d2f98b36720gb3cd78241f6b0g4a173edg8bc754g801cd7963bg549f6c8214dg0ca94d5601ceb8g4a9e3b0d7691450fe91ab5602gd49fe28g51cbe69a532e6fgab57419e327d4a60g2bef9a

**(11,6,25):** 01234567018395a7123456701

**(12,6,33):** 01234567018926a317b96435709812347

**(13,6,47):** 012345607183940a1b35290cb6579140ba7831052a76190

**(14,6,66):** 012345607183940a12354b061c9723068db725610978a564b972305698170ca629

**(15,6,89):** 012345607182930a142b35061743c085d1230469157038a1e743bad06738bcd2a961cb475869ac037892abdc8

**(16,6,117):** 012345607182930a142b35061743c085d1230469157038a1420e361f25073b124086915e0342cbdea50319845b3d9607132489e012cd9ab4e0235

**(17,6,200):** 458g73bc40g8ef190d45ea9178b2a6ef732b014c3g780dc4abe5d901a65e347f62ab3gf7de08gc34d98067a195de621ag03b2f67gcb39ad4c8g0954d236e519a2fe6cdg7fb23c87g569084cd5109fg2a1e56fba289c3b7fg843c125d40891ed5bcf6ea9b

**(13,7,31):** 0123456708294a6b01c923456709182

**(14,7,42):** 0123456708914a6b03c94567d32a9417803ab46578

**(15,7,55):** 0123456708194a2b031c4562081dce597821bc39a4865b7901283ba

**(16,7,72):** 0123456708192a3b041c253d06172e3f094a217c59ba813cd49021bc56749adbce35421

**(17,7,98):** 041c253d06172e48031f259406172a3b051g789dc0b324ed109ab76de2c0153be89a0124cefd3b072e95dg0123456c081g