

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

SIMON HOOD\*

DANIEL RECOSKIE†

JOE SAWADA‡

CHI-HIM WONG§

**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 maximal. 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.

**Index Terms**—snake, coil, circuit code, single-track, snake-in-the-box, longest path.

## I. 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 [21], [8], [33], [13], [24], [30], [25], [3], [37], [31], [36], [4], [38], [31], [5], [20], [11], [18], [28], [7], [9], [12], [16], [19], [23], [22], [27], [32], [35], [42], [6], [41] including many dedicated to improving bounds [10], [1], [39], [2], [34], [14], [40], [15], [26]. They were first described by W.H. Kautz [21] in relation to a theory of error correcting codes, but since have appeared in many other applications including electrical engineering, coding theory, combination locking, and network topology. Generally, the longer

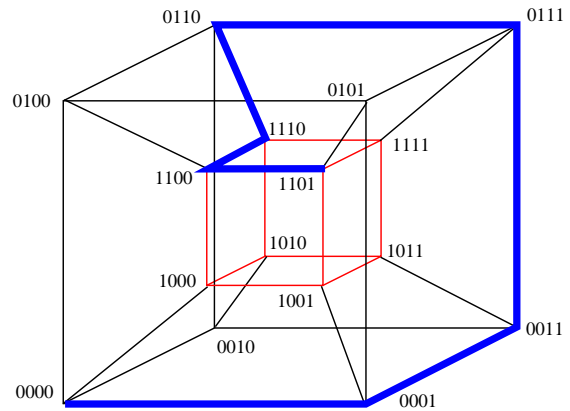


Fig. 1. A maximal snake in  $Q_4$ : [0000, 0001, 0011, 0111, 0110, 1110, 1100, 1101].

the snake or coil, the more useful it is [24]. Finding the longest induced paths, however, is well known to be NP-hard even when graphs are bipartite [17]. The complexity is unknown if the graphs are restricted to hypercubes [31].

The  $n$ -dimensional hypercube, denoted by  $Q_n$ , 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. An example of a maximal snake in  $Q_4$  is illustrated in Figure I.

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.

\*Research supported by NSERC. email: simon.hood@gmail.com

†School of Computer Science, University of Guelph, Canada. Research supported by NSERC. email: drecoski@uoguelph.ca

‡School of Computer Science, University of Guelph, Canada. Research supported by NSERC. email: jsawada@uoguelph.ca

§School of Computer Science, University of Guelph, Canada. email: cwong@uoguelph.ca

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 maximal  $k$ -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 R.C. Singleton [33] using explicit constructions. Subsequently, Klee [23], [22] 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 [9], Paterson and Tuliani [28], Hiltgen and Paterson [18], Zinvoik et.al [42], and Chebiryak and Kroening[6]. In [18], 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 adaption of Kochut's [25] 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 maximal. 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 [37], [4], [5], [11], [30], [3].

In addition to exhaustive search, we also applied some optimizations to a *single-track circuit code* construction algorithm by Paterson and Tuliani [28]. 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 I provides the lengths

of the longest known  $k$ -coils and Table II 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 [28], [42], [6], [29].

The remainder of the paper is outlined as follows. In Section II we outline two search algorithms with a comparative analysis. In Section III we outline a variety of optimizations for  $k$ -coils and provide an experimental analysis. In Section IV we introduce single-track circuit codes and include an updated table of the longest known such codes. A summary is provided in Section V. 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 <sup>1</sup>.

## II. EXHAUSTIVE SEARCH ALGORITHMS

Kochut's algorithm [25] that exhaustively searches for maximal 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 will 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 less than  $k$  bit positions.

<sup>1</sup>This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARC-NET:www.sharcnet.ca) and Compute/Calcul Canada.

TABLE I  
LONGEST KNOWN  $k$ -COILS. THE NUMBERS IN PARENTHESES REPRESENT BEST KNOWN VALUES  
PREVIOUS TO THIS RESEARCH.

$n$	$k$					
	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	170	64 <sup>a</sup> (56)	30*	24*	18*	18*
10	340	102 <sup>a</sup> (86)	46*	28*	20*	20*
11	620	154	70 <sup>a</sup> (68)	40*	30*	22*
12	1238	288	102 <sup>a</sup> (98)	60 <sup>a</sup> (56)	36*	32*
13	2468	494 <sup>b</sup> (442)	182	80 <sup>a</sup> (78)	50*	36*
14	4934	812 <sup>b</sup> (700)	280	106 <sup>a</sup> (102)	68 <sup>a</sup> (66)	48*
15	9868	1380 <sup>b</sup> (1290)	480 <sup>b</sup> (450)	210	88 <sup>a</sup> (82)	60 <sup>a</sup> (58)
16	19740	2240 <sup>b</sup> (2176)	768 <sup>b</sup> (672)	288	118 <sup>a</sup> (106)	76 <sup>a</sup> (72)
17	39840	3910 <sup>b</sup> (3842)	1224 <sup>b</sup> (1088)	476	204	102 <sup>a</sup> (90)

\* value known to be optimal

<sup>a</sup> found by partial exhaustive search

<sup>b</sup> found by optimized implementation of construction from [28]

TABLE II  
LONGEST KNOWN  $k$ -SNAKES. THE NUMBERS IN PARENTHESES REPRESENT BEST KNOWN VALUES  
PREVIOUS TO THIS RESEARCH.

$n$	$k$					
	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 <sup>*a</sup> (33)	19*	11*	10*	9*
9	188	63 <sup>a</sup> (53)	28 <sup>*a</sup> (26)	19*	12*	11*
10	363	103 <sup>b</sup> (83)	47 <sup>*a</sup> (42)	25 <sup>*a</sup> (23)	15*	13*
11	680	157 <sup>b</sup> (151)	68 <sup>a</sup> (64)	39 <sup>*a</sup> (35)	25 <sup>*a</sup> (24)	15*
12	1260	286 <sup>b</sup> (285)	104 <sup>a</sup> (94)	56 <sup>a</sup> (51)	33 <sup>*a</sup> (30)	25*
13	2466	493 <sup>b</sup> (439)	181 <sup>b</sup> (178)	79 <sup>a</sup> (73)	47 <sup>a</sup> (44)	31 <sup>*a</sup> (29)
14	4932	811 <sup>b</sup> (697)	279 <sup>b</sup> (276)	112 <sup>a</sup> (97)	66 <sup>a</sup> (60)	42 <sup>a</sup> (41)
15	9866	1379 <sup>b</sup> (1287)	480 <sup>b</sup> (446)	206 <sup>b</sup> (205)	89 <sup>a</sup> (76)	55 <sup>a</sup> (51)
16	19738	2240 <sup>b</sup> (2173)	766 <sup>b</sup> (668)	285 <sup>b</sup> (283)	117 <sup>a</sup> (100)	72 <sup>a</sup> (65)
17	39838	3941 <sup>b</sup> (3839)	1223 <sup>b</sup> (1084)	473 <sup>b</sup> (471)	200 <sup>b</sup> (198)	98 <sup>b</sup> (83)

\* value known to be optimal

<sup>a</sup> found by partial exhaustive search

<sup>b</sup> found by starting from a longest known coil and backtracking

```

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

```

Fig. 2. Exhaustive search algorithm for maximal  $k$ -snakes or  $k$ -coils in  $Q_n$ .

### A. Approach one: 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  $c = \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{k-1}$  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 **Search**( $t, d$ ) in Figure 2. The  $k$ -snake ( $k$ -coil) being searched at the start of each recursive call is stored in  $\alpha = a_0 \dots a_{t-1}$ , where each  $a_i$  is the integer representation of a vertex. 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,
- $conflict[x]$ : a set of the  $c$  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  $0, 2^1 - 1, \dots, 2^k - 1$ , incrementing 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  $a_0 \dots a_t$  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 less 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 less bit set to 1. Using these data structures, the function **CheckMaximal**( $t$ ) shown in Figure 3 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 less 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  $a_0 \dots a_t$  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**( $t, d$ ).

```

function CheckMaximal(t: int) returns boolean
int x
1: if bits[at] > k or maxlen ≥ t+k then return FALSE
2: if bits[at] = 1 then
3:   Print(a0 ··· at)
4:   maxlen := t + 1
5:   return TRUE
6: for each x ∈ adjSmaller[at] do
7:   at+1 := x
8:   if numConflicts[x] = k and CheckMaximal(t + 1)
   then return TRUE
9: return FALSE

```

Fig. 3. Recursive function to check if an initial search path  $a_0 \cdots a_t$  can be extended to a  $k$ -coil.

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.

### B. 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 maximal  $k$ -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

```

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

```

```

procedure Search2(t, d: int)
int i
1: for i from 1 to Min(d + 1, n) do
2:   at := adj[at-1][i]
3:   if not IsConflict(t, 0) then
4:     CheckMaximal(t)
5:     Search2(t + 1, Max(i, d))

```

Fig. 4. Alternate search algorithm for  $k$ -snakes and  $k$ -coils in  $Q_n$ .

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 4. 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( $t+1, k - \text{bits}[a[t]] + 1$ ) 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.

### C. 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)
<b>Search</b>	680210	137	5.04	1.02	23606	2009
<b>Search2</b>	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  $2c = 2\binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{k-1}$  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** will always be faster than **Search2**. For  $k = 3$ , **Search2** will be faster when  $n \leq 6$ . For  $k = 4$ , **Search2** will be faster for all  $n < 12$ . For  $n = 12$ , it will be 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** will be faster for  $n \leq 14$ .

### III. 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]*.

#### A. Connectivity

In this subsection we consider a simple connectivity property when searching for maximal  $k$ -coils. In particular, given a  $k$ -snake  $\alpha = a_0 \dots a_t$  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 a  $O(2^n)$  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  $V_0 \dots V_n$  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 maximal  $k$ -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)
<b>Non-optimized</b>	651075	26060	556120186	4624	7817954	1963	557993
<b>Connectivity</b>	389257	15719	300717441	2424	3840003	1345	277788
<b>% Reduction</b>	40	40	46	48	51	31	50

It does not appear 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.

#### B. 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 5. Depending on which of the 8 vertices is the starting vertex, up to 8 different equivalent vertex sequences can be obtained, even when the vertices are relabeled so they match the canonical form outlined earlier: label the first vertex  $0^n$  and visit

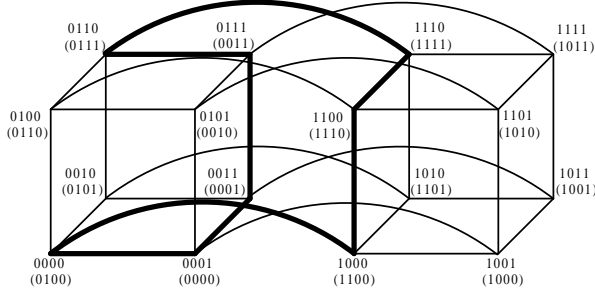


Fig. 5. Illustration of two rotationally equivalent 2-coil sequences in  $Q_4$ .

dimension  $d$  before  $d+1$ . The two different, but equivalent 2-coils illustrated in this figure are listed below:

[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  $O(t) = O(\text{snakelength})$ .

In Section III-B5, we provide an analysis comparing the impact on the number of recursive calls and the overall run time for feasible values of  $n$  and  $k$ .

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 com-

plements 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.

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  $pdl\text{list}[v]$  for each vertex  $v$  that contains the  $n$  vertices that are pre-diagonal to  $v$ .
2. Maintain global the 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  $v \in pdl\text{list}[a_t]$ .

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) *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  $v \in pdl\text{list}[a_t]$  where  $b[v] = \text{TRUE}$  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  $v \in pdl\text{list}[a_t]$  such that  $t - pos[v] = len$ . The flag  $f_2$  is TRUE if and only if there exists a  $v \in pdl\text{list}[a_t]$  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.

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 6. 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.

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 the rotational testing. Also, even though they are illustrated in the table, we do not need to consider the bit count sequences starting from  $a_t, a_{t-1}, \dots, a_{t-k}$ , as they will always start  $0, 1, \dots, k$ .

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

1. Pre-compute the hamming distance between ever 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 test-

$\alpha$	00000	00001	00011	00111	01111	01101	11101	11001	11011	11010	10010
00000	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>3</b>	<b>4</b>	<b>3</b>	<b>4</b>	<b>3</b>	<b>2</b>
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		
<b>01101</b>						<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>
11101							0	1	2	3	4
11001								0	1	2	3
11011									0	1	2
11010										0	1
10010											0

Fig. 6. A dead end search path  $\alpha$  when considering rotational equivalence using the bit count sequence.

ing. 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.

5) *Experimental results:* In Table III, the number of recursive calls required by the non-optimized search algorithm 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. Note that both search algorithms **Search** and **Search2** will require the same number of recursive calls in each case, but the time to generate each recursive call will vary.

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 IV  
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).

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

Table IV. 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.

### C. Applying the optimized algorithms

Using the optimized exhaustive search algorithm, we used SHARCNET<sup>2</sup> 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, and verify several previously known long  $k$ -snakes and  $k$ -coils were indeed maximal. These results are shown in Table I and Table II. Instances of each new longest  $k$ -coil and  $k$ -snake are given in the Appendix.

<sup>2</sup>This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARC-NET:www.sharcnet.ca) and Compute/Calcul Canada.

TABLE III  
A COMPARISON ON THE NUMBER OF RECURSIVE CALLS REQUIRED FOR EXHAUSTIVE SEARCH ON SELECT  $k$ -COILS.

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

#### IV. 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, the first track in the 2-coil [0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000] is the sequence of the bits from the first 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 the first track and so the 2-coil is also a single-track circuit code. As another example, consider the 2-coil [0000, 0001, 0011, 0111, 1111, 1011, 1010, 1000]. 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 [19] with a focus on circuit codes in [18].

Many of the previously known longest  $k$ -coils are due to a construction for single-track circuit codes by Paterson and Tuliani [28]. 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 V. 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 I and Table II respectively.

#### V. SUMMARY

By optimizing an exhaustive search algorithm, we find 13 new longest  $k$ -coils, 21 new longest  $k$ -snakes, and

TABLE V  
LONGEST KNOWN SINGLE-TRACK CIRCUIT CODES. THE NUMBERS IN PARENTHESES REPRESENT BEST KNOWN VALUES PREVIOUS TO THIS RESEARCH.

$n$	$k$					
	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 known to be optimal

verify that some of them are maximal. By optimizing an algorithm by Hiltgen and Paterson [28] 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] 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.
- [5] 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.
- [6] Y. Chebiryak and D. Kroening. An efficient SAT encoding of circuit codes. In *International Symposium on Information Theory and Its Applications*, 2008.
- [7] 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.
- [8] D. Davies. Longest ‘seperated’ paths and loops in an  $n$  cube. *IEEE Transactions on Eletronic Computers*, page 261, 1965.
- [9] K. Deimer. Some new bounds for the maximum length of circuit codes. *IEEE Transactions on Information Theory*, 30(5):754–756, 1984.
- [10] K. Deimer. A new upper bound on the length of snakes. *Combinatorica*, 5:109–120, 1985.
- [11] 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.
- [12] 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.
- [13] R. J. Douglas. Upper bounds on lengths of circuits of even spread in the  $d$ -cube. *Journal of Combinatorics Theory*, pages 206–214, 1969.
- [14] 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.
- [15] P.G. Emelyanov and Agung. Lukito. On the maximal length of a snake in hypercubes of small dimension. *Discrete Mathematics*, 218(1–3):51–59, 2000.
- [16] T. Etzion and K. G. Paterson. Near-optimal single-track Gray codes. *IEEE Transactions on Information Theory*, 42:779–789, 1996.
- [17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman San Francisco, 1979.
- [18] A. P. Hiltgen and K. G. Paterson. Single-track circuit codes. *IEEE Transactions on Information Theory*, 47(6):2587–2595, 2001.
- [19] A. P. Hiltgen, K. G. Paterson, and M. Brandestini. Single-track Gray codes. *IEEE Transactions on Information Theory*, 42:1555–1561, 1996.
- [20] 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.
- [21] W. H. Kautz. Unit-distance error-checking codes. *IRE Trans Electronic Computers*, pages 179–180, 1958.
- [22] V. Klee. A method for constructing circuit codes. *Journal of the Association for Computing Machinery*, 14:520–528, 1967.
- [23] 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.
- [24] V. Klee. What is the maximum length of a  $d$ -dimensional snake? *American Mathematics Monthly*, pages 63–65, 1970.
- [25] K. J. Kochut. Snake-in-the-box codes for dimension 7. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 20:175–185, 1996.
- [26] A. Lukito. An upper bound for the length of snake-in-the-box codes. In *6th International Workshop Algebraic and Combinatorial Coding Theory*, 1998.
- [27] 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.
- [28] K.G. Paterson and J. Tuliani. Some new circuit codes. *IEEE Transactions on Information Theory*, 44(3):1305–1309, 1998.
- [29] W. Potter. Latest records for the snake-in-the-box problem. In <http://www.ai.uga.edu/sib/records/>, 2011.
- [30] 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.
- [31] 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.
- [32] M. Schwartz and T. Etzion. The structure of single-track Gray codes. *IEEE Transactions on Information Theory*, 45:2383–2396, 1999.
- [33] R. C. Singleton. Generalized snake-in-the-box codes. *IEEE Transactions on Electronic Computers*, pages 596–602, 1966.
- [34] H.S. Snevily. The snake-in-the-box problem: A new upper bound. *Discrete Mathematics*, 133(3):307–314, 1994.
- [35] 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.
- [36] C. A. Taylor. A comprehensive framework for the snake-in-the-box problem. *Thesis for Master of Science, The University of Georgia*, 1998.
- [37] 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.
- [38] 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.
- [39] J. Wojciechowski. A new lower bound for snake-in-the-box codes. *Combinatorica*, 9:91–99, 1989.
- [40] G. Zémor. An upper bound on the size of the snake-in-the-box. *Combinatorica*, 17:287–298, 1997.
- [41] 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.
- [42] 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.

## APPENDIX A

INSTANCES OF LONGEST  $k$ -COILS AND  $k$ -SNAKES

Each  $k$ -coil and  $k$ -snake can be represented compactly by its bit-change (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 bit-change sequences for each new longest



70456f706b3f3gc37019ed87g08bc1d6gdb0839cdf6135f73ag7eaf69ed8fa307daefdc  
1624a19cd5fb1g0c23a19fcdg27c8abed9abc7eb60ag1ed4625c04753149723cb781  
369457f8357ed0cb61ge584e3dc76835ac348fd04a1ef30245d2b40g7ea6cba9fe4a7  
f3bc817f89af1bg34593e5b926780928d501e5923bgfa912ade3f81df2a5bef083570  
95c19gc08bgf0c529f9c0fe3846c3bef70d312e45c3b0ef149eacd9fbcde9d82c13g  
f6847e2697536b945ed9a358b6790a579gfd2ed831e7a6g5fe98a57ce56af026c3g052  
467f4d6219gc8edcb0g6c905dea390abc03d1567b5g7db489a2b4af723g7b45d10cb  
34c9g50a3f04c7d02a5792b7e3b1e2ad10c2e74b0e120g5a68e5dg092f534g67e5d  
2g036bgcef10defgfb1fa4e35108a69g48b9758db67gfb57ad89b2c79b104gfa53g9c  
8170gbac79eg78c0248e5127468906f843b1eagfed218eb27fgc5b2cde25f3789d719  
fd6abc4d6c094519d6f732ed56e0172c5026e9f124c79b4d9g5d3g4cf32e4g96d2g3  
4217c8ag7

**(11,4,70):** 012345617358691527610a2860472631456074852937625a98231780a9  
246570a4873

**(12,4,102):** 0123456137580635209165071a630126504176081265092317b024716  
8249a1736259712067329876124a6531a76925a614ba

**(15,4,480):** 01234567895a70628ba653b89c4bd1208db75309ce31980bc610376ce  
4562db8c269378e4a7dec8640d87904a530b26c4b0e79ca5192a4c0582c9e851378  
6b04568a9e413deb15483cb4eac3d79c068530c1ea5d72a6d35c7465a14729e480c3  
784da1329b10273495031d59bea5c8479c521d7be6d8b975e387d23e6a134c59e43  
e2f03a5f0e1d32b1c4f08d7c6851de364c1a5701e8db2e6f4cd083cadf520495a6cd  
721ab7352f0b6a5831d5f7294fbc6a2d70a82c34d6e38ba2145891034cd9b8370523  
c146ec9ab8421d874a062bf079845637e5d06a2e9701d340a56fbae89764527168db  
49cd1e763b01f32db84fe1d5206d83bc98f7e1b63415b7296ae25f1b09d5c042976cf  
5234db2709ae7c1f59b0653914ebf843c59de23ad64e1bac34062941de8f1a5c3e9d  
b30e56f9c760a3e2f4082b6f598a06db4e652f7c583a0fe290df3bcea1bd80f4c6d749  
bc3e78db296fb34c1a3708dcf4ed2c09af85927dcfab216e9a0f17294ebc906a5801d  
72accf24ade8c73e412ab8945bfe8dc5146e9aedb837d52148abc4682f7a10f65489  
7e639cf72a356fbc8e2f9

**(16,4,768):** 0123456789a6b74c185dc367e1fb0eac14803bc9af7c4e1d54063b17e  
8b916a5732a90b1f5c9df8a567d09ae8c1a6f5236db0951f79e5b831048ed95c3ae2c  
783b12de8f7a58bc340b29de35c1ef39705d67f2e3a08f4a1709542f7c18379a06d94  
e2f03a5f0e1d32b1c4f08d7c6851de364c1a5701e8db2e6f4cd083cadf520495a6cd  
721ab7352f0b6a5831d5f7294fbc6a2d70a82c34d6e38ba2145891034cd9b8370523  
c146ec9ab8421d874a062bf079845637e5d06a2e9701d340a56fbae89764527168db  
49cd1e763b01f32db84fe1d5206d83bc98f7e1b63415b7296ae25f1b09d5c042976cf  
5234db2709ae7c1f59b0653914ebf843c59de23ad64e1bac34062941de8f1a5c3e9d  
b30e56f9c760a3e2f4082b6f598a06db4e652f7c583a0fe290df3bcea1bd80f4c6d749  
bc3e78db296fb34c1a3708dcf4ed2c09af85927dcfab216e9a0f17294ebc906a5801d  
72accf24ade8c73e412ab8945bfe8dc5146e9aedb837d52148abc4682f7a10f65489  
7e639cf72a356fbc8e2f9

**(17,4,1224):** 0123456789abc8de341b8c9e6d35c01a93f102ca850cd3g198da3f890  
714g2ef0396201748ac67d235e9g310c6893g50d1c7g4e85cb84age374g1efc8531ec  
b354286fab0b4c5da48263eged21ac705fc84g35cf1748g2f6307g936ef0c26f8gb37c  
80g9c76a3d8e496g71e63adcf01a8eg247bg36f1c7gb2683fabdc42f5cd0b4gadbf3f9  
c2g34f5g2dec19065df280dce1g4b8e30fa629fcb8g2f9ad3cbe91g6ab7g11496fe19  
cb5gaf6cb7fa10g854d71ba341g08f6930c4bedae5bg193fab5e1cg9058fde92f865db  
085g97f7ebgd2be84f3761289ec68f43bd5c4g6901e79f85cbe9708gff5473b105ab3d  
719437f52b09f15a9036bc2d8a350gd3b6c917g6fd548025b37g905243fb762c9847  
e9c12856c2b7a945b87e54cd9ga13ec74f1c9dg582fdb17634a79c2f547a6cb92dag5  
36205g8a37dga92e567932070g15fe8c0g26b8g51f73ab1982dc6e25gab762ged95a  
1ef7cda47f3ec21fe5a07d25ca42df87b03g4fad93f78b2ce9853a1gd0a7fe92da01f5  
7e80b2g1e62bc0ga8b07e421a7ge6a1b3294cf6be15cb239ag0537ce8f14e2b05a1e  
48b720349af80da9g4fe394206a8e2f0de89ca56gbd9087g9ac5ef47c2g03b860a947  
e806392a4c65eb341e5f6b0c56a4de30ab41035gef7d915432f5eg70b62gaf4c93d4e  
562034dc5ae6gd709c6807bd94g7de610c4e9684c7f021b5876cab70f249dafeb6g5  
c1607da4c61g7e0df1245gd3429156f210d84g605d36g2b4a89732d9e924ba651eb  
09df7g8d421e6gd8f2041b8a67f1ca6587dba84136fd471cdfa96c352ca1f05a69ed7  
809451b2f316a80d13ba46893cd2b8gde73219e368cdeb1628g1be5d0c7age8b847ed

**(12,5,60):** 012345678239a562b9145678910a7631204567820ba5629b145678b13a  
7

**(13,5,80):** 012345671824951a640253a4b0c382401932a709b2640719cb3754192a  
38c12097a14230b762a3c

**(14,5,106):** 012345671839501234560a2b35c012345608319507329ad7062ca7312  
5a43085942635104c36a0b423506143870c1359721ca569d

**(14,6,68):** 01234567829a45b702c4adb301ca9b7601892570cd12473ad82970ab68  
475d3a64c

**(15,6,88):** 012345607182930a142b35061743c085d123046915703e9ca10426bc13  
4508a9c765421cabd76350417c8b5

**(16,6,118):** 012345607182930a142b35061743c085d1230469157038a1420e361f2  
5073b12408691370d582a74019e3876dc210e67b319a246735d0a19f7b5a

**(16,7,76):** 0123456708192a3b04c1253d06172e480597bf18c4a92307c86e24b5980  
12cea4870d62afb7

**(17,7,102):** 0123456708192a3b04c1253d06172e48031f259406172a3b051g789dc  
0b324ed109ab76de2c0153be89a0124cefdb3072e95d

**(15,7,60):** 2e5f1b9ac682531946cd5e17402cb6184d9c2e1a5d8327eb1c682470bd9

*B. k-snakes*

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

**(9,3,63):** 0123042501230627041354230413562805240321052407265314032453  
14072

**(10,3,103):** 15361234153012391562413501836123450237153612341530123914  
37153012341536123715381239153012341536123915302

**(11,3,157):** 012341563078209867a52795831503786218023942598371435190a1  
203984a924056415903a601a527a42059675462186a1520782a7143764215831684  
a987a14239473a603864a190a89675937a

**(12,3,286):** 012345670829710947830912087901465a7b685b06a95b75a6b58041  
3624a12543b12a613421a50894703871092873489078315ab0659a685b7a690ab56  
a98132541b34a12634b532143ba897108290387490219780923ab685a7b59a60b57  
8b6a5b79324a13621b345216a243126b970389478290178437098742b659ab06a7  
b586a0965ba607241b32543621a43

**(13,3,493):** 0123456705189ab147892460c4a9c579417c640c7814ca54b2063b1a2  
071c5916094a01ba9c159a2b19641375c83b675ab940bc50165b3609b40673b0c1b  
8a49283ca4630153945bc438c50315ca8359b32610728961c85b48014391829458b  
4962840387cb5a720cb9243125b180b270142310c721582a9346a7590718b743b2  
537a5b178b59a7b4276081c6a4085ab23a183748a643ba23406a317ac52b9c61524  
6378b628a126c18367815c68ba6947309cb471c8a2c3726b7c9b28ca2b49c236c01  
a850931ab926798a7c3a90372967310978c95b624508b6307ca026a986058a70ca8  
b50a29043c714523c85a9657c602c5426a

**(14,3,812):** 012345678965127061a82b9164cd568ab083951a640d719084c298c16  
0c8ba327901725b9d7580bc1573462708cd0a125873d6951d034b10457d40c8ab91  
d59b2c16920dc4529a37b9d046d85b209a671256da3c5d32963d408c15621cb4571  
bd6432b18a9c16d37602cbd18795b2768a426ab17a6d304527b54c3295c673abc50  
814576a97db4c650912cb97083b78c5987a6d32b9c234ab12479a8c42d053297819  
6c3472d15b4c19d0ac9042109876abc14ba38c5b3918043b6d2ab1905174a39b652  
c34516d841d3b5d10978c453c8a042ca150d3ac76b8c51d25938a1c72b4a3257603  
662565d190432a408d3b4852d6a8497c04256b21a08549bc38ab297da2784b7265  
1d3ab83d06ac302b67803194d3b27cb58d0231c4a08cb1968b903c9b7256a8c0af6d  
784adb790da5136ac94c206dba5438d04c5170c1da41c9b27804d87690386c491  
d825a784c134bd76c823a06d34259d45683541cb90d360971da074315670b28903  
45a3c69740ba8d76a3b2163270a2354c1d6a7d19568d93a5279dcb01da328a47193  
dc806978abc57ab9d

**(15,3,1380):** 012345674891a48bc69b2d74a6b5ad35014e8b0e5c8d506c5168ab20  
eda756a913e5b409d347d0e326158ac1928cb9e4dc82da091da36803c72d183c174c  
b9d5a3b56a7cb86c98a13eb5712c810945c3db074d27b54e89ca1690e6a05d76ae  
71b097148ab452e79a4692d6307c143c681263a860a19453c29e6a9b0dc6473b2d7  
e23cd5a061980b5184bc72815293b029da13dce5201d80e784b269d468a9e841a8b  
190dc46e058103b768d243e725e467c1b890ab3c9ad362e9c043be07194765ceb9  
7ab52ad3e807d8a105ad91a390b76d85bca9b4328a7ed452ce5d82693a0b1346017  
48e51065bd435b290d28c6530213ce1745ab27a19bc1709140b3287ac36103d4ea1  
257dc56c7ae8041b394d8b92da5c9b8c37d4c3e0b7ea68c4be946592dc13e219036  
92b09db34ea216489b47d519ec2765c86215abd9340d7a30e71c603a6427d645b32  
518a6d350d80c7e6945e90b480e3b0734d51e98da03d27c9056e28c6a8e9c13704d  
b7214b52968b418de278dc34ec9a1874cb7a6b5280dc50b3dab543b247c950a71b  
47c260bc85ea681a506942bd732e9d3ce08a3d9a75e2a764d56019a2d632183ceab7  
6cb34713cd43ed7260cb1293d25e8b36ac158a91cb80de3724e507465ba147012c5  
e128d7c8b901e784e9a4651328634d29467d4572e8b639e047ec5a34816c9a10963  
ab7542ed5cb2d8c19d2b9e6c59ea726a30b952ad501d8c94ea84d7e0d827dc2e5a3  
8405bd256c14da986019b084132cde57c63e7a64907e30586c0512e814b30ce17cb  
97a60d51ad725b7ae276e5c14adbc37ec869d710a8b93bd9a4e6526845218d0b



8e54f96c37a9g146251d840ag9127d53f72g05fc1e73c41gd523g1ce5f682c7bga58f  
7cb32796ea54g1ef89ab78039c1fe8bg0746gbfc76582g4598f07b4f85276a3b5gdf1  
736g5d4bgea2179f8263e1dg3c4e58623dfcg9afd65ga73bf97e36cg9637bga14d7f  
068g4af709df21b8ge63ba4280f459273ab4065fe160a7f1g4d6eg24a5f0ea4gdf189  
0g6ca3f1g6cce06b8d3f2a4d19b3c697ebg41d9ca7628ac1g68f90a2fb9176c219f06  
83ecfa5146e8af52cad3046b1908ed45aeg2df980e51gab3158fa36ec1b6de8ga5b8

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

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

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

**(13,5,79):** 01234506172850913254061a2b03172c045b239654013894ba13670943  
5a6c29173ca897064a8b5

**(14,5,112):** 01234506173850912340561a2073b124053812c063192704135780a12  
73d069581c63052469850c73695b814ac98523c4795086391428596

**(15,5,206):** 7bde5a0b792e3b0467d38402a7b48ce06b1c8a304c1578e49513b8c59d  
017c2d9b415d268905a624c9d6ae128d3eac526e379a16b735dae7b0239e40bd637  
048ab27c846eb08c134a051ce748159bc38d9570c19d245b162d085926acd49ea68  
1d2ae356c273e4

**(16,5,285):** 94f7cd02e8a36c05b72d80569b713cf59e40b619ef240ac58e27d94fa27  
8bd935e17b062d83b01462ce7a049fb61c49adfb57039d284fa5d23684e09c26b1d8  
3e6bcf1d7925bf4a61c7f458a602be48d3fa508de13f9b47d16c83e9167ac824d06af  
51c72af0351bd69f38ea50b389cea46f28c173e94c12573df8b15a0c72d5abe0c6814  
ae3950b6c34795f1ad48

**(17,5,473):** 34fgac96e287c19fb547g23fdc7g8934f0eb27dc06e3ga9c478310c4de8  
9352g7c105b284fe09cd8650912de8a74c065ag7d9325e01dba5e6712dfc905baf4c  
1e87a2561gfa2bc67130e5agf39062dcf7ab643f7g0bc6852af438e5b7103crgb983c  
45g0bda7f398d2agc658034ged809a45g1fc38ed17f40bad589421d5ef9a46308d21  
6c395gflade9761a23ef9b85d176b08ea436f12ecb6f7823egda16cbg5d2f98b36720  
gb3cd78241f6b0g4a173edg8bc754g801cd7963bg549f6c8214dg0ca94d5601ceb8g  
4a9c3b0d7691450fe91ab5602gd49fe28g51cbe69a532e6fgab57419e327d4a60g2b  
ef9a

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

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

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

**(14,6,66):** 012345607183940a12354b061c9723068db725610978a564b972305698  
170ca629

**(15,6,89):** 012345607182930a142b35061743c085d1230469157038a1e743bad067  
38bcd2a961cb475869ac037892abcd8

**(16,6,117):** 012345607182930a142b35061743c085d1230469157038a1420e361f2  
5073b124086915e0342cbdea50319845b3d9607132489e012cd9ab4e0235

**(17,6,200):** 458g73bc40g8ef190d45ea9178b2a6ef732b014c3g780dc4abe5d901a6  
5e347f62ab3gf7de08gc34d98067a195de621ag03b2f67gcb39ad4c8g0954d236e51  
9a2f6cdg7fb23c87g569084cd5109fg2a1e56fba289c3b7fg843c125d40891ed5bcf  
6ea9b

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

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

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

**(16,7,72):** 0123456708192a3b041c253d06172e3f094a217c59ba813cd49021bc567  
49adbce35421

**(17,7,98):** 041c253d06172e48031f259406172a3b051g789dc0b324ed109ab76de2c  
0153be89a0124cefd3b072e95dg0123456c081g