The Trapped Knight

Alexander Rao, alexhrao@gmail.com

July 15, 2020

Abstract

The Trapped Knight has been studied in detail by many mathematicians, most notably Neil Sloane. However, little has been done to clearly model the knight's journey. In this paper, we will explain the problem, show how it can be efficiently modeled, and present an example implementation.

1 Background

The Trapped Knight is a problem concerning an infinitely-sized chess board. Each square is numbered in the pattern of a "square spiral". The first square is numbered 0; the square below it is numbered 1. From there, the squares are numbered in a counter-clockwise fashion.

42	41	40	39	38	37	36
43	20	19	18	17	16	35
44	21	6	5	4	15	34
45	22	7	0	3	14	33
46	23	8	1	2	13	32
47	24	9	10	11	12	31
48	25	26	27	28	29	30

Table 1: Example 7x7 Trapped Knight Board

Difference from Canonical Problem The problem, as presented by Neil Sloane, uses **1** as the starting index, not 0. However, using 0 as the first index makes it far easier to recognize the repeating patterns within the square spiral. Should compatibility with Neil Sloane's version be of utmost importance, minor changes can ensure this. We shall notate exactly what must change throughout this document. Table 2 depicts the board for Sloane's version.

43	42	41	40	39	38	37
44	21	20	19	18	17	36
45	22	7	6	5	16	35
46	23	8	1	4	15	34
47	24	9	2	3	14	33
48	25	10	11	12	13	32
49	26	27	28	29	30	31

Table 2: Sloane's 7x7 board

The knight begins in the center and moves according to the following rules:

1. It must move in the shape of an "L" — 2 units along an axis, then 1 unit along the other axis. For example, a knight could move down 2 rows, then left (or right) 1 column. A knight could also move left 2 columns, then up (or down) 1 row.
2. It cannot visit a square it has previously visited
3. It must move to the smallest viable square

42	41	40	39	38	37	36
43	20	19	18	17	16	35
44	21	~~6~~	~~5~~	~~4~~	15	34
45	22	~~7~~	~~0~~	~~3~~	14	33
46	23	~~8~~	~~1~~	~~2~~	13	32
47	24	~~9~~	10	11	12	31
48	25	26	27	28	29	30

Table 3: Example Trapped Knight Board, after 10 moves. Note the knight is on index 7

The first 10 moves are: [0, 9, 2, 5, 8, 3, 6, 1, 4, 7]
From here, the knight would move to 10.

2 Definitions

Given that the first index is in the center of the board, using traditional indexing (i.e., top left being 0, one to the right is 1, and so on) is ill-suited for this problem. Instead, we will use 2-dimensional indexing, where "rows" are the vertical offset from 0, and "columns" are the horizontal offset from 0. A positive row indicates a row above 0; a negative row indicates one below. Similarly, a positive column indicates a column to the right of 0; a negative column indicates a column to the left of 0.

2.1 Indices and Subscripts

A single chess square can be represented in two different ways — either its row and column, or its index within the spiral.

- A square's **subscript** is a tuple of its (row, column).
- A square's **index** is the number on the square; i.e., (-1, -1) is 8.

As shall be shown later, there is a unique and calculable relationship between these two representations, allowing them to be used interchangeably.

2.2 Frames

A **frame** is the name for a complete square ring. Frame 0 is defined as the center square.

A frame is defined in a similar fashion to a circle; an example has been shown in figure 1. It is the collection of all squares that are equidistant from the center. Note that the distance to the center is the maximum of the absolute value of a square's row and column. For example, the distance of square (-2, 1) is 2. Formally, it can be said:

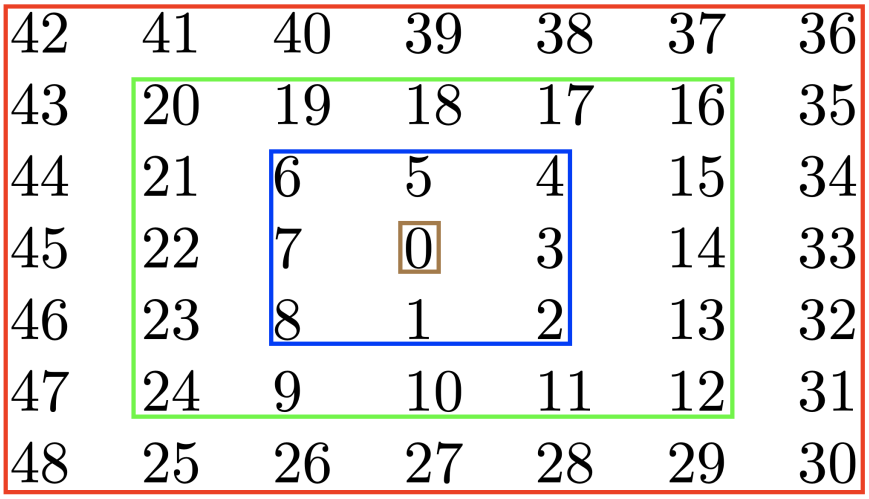

Figure 1: Trapped knight board, frames f_0 through f_3

$$D = max(\mid row \mid, \mid column \mid) \tag{1}$$

For ease of discussion, we define the following properties of frames:

- A frame's **index** is the "radius" for that frame: for example, the square (-2, 1) is in the frame with index 2. A frame is denoted as F_f, where f is the zero-based index.
- A frame's **extremum** is the value of its bottom-left square. For example, the extremum of f_2 is 8. The extremum is always the largest index within the entire frame. Denoted as F_e
- A frame's **base value** is the value of the square immediately to the right of the frame's extremum. The base value is always the smallest index within the entire frame, and is always immediately below the previous frame's extremum. Denoted as F_b
- A frame's **tag** is the value of the top right corner of the frame. Denoted as F_t
- A frame's **perimeter** is the number of squares that make up the frame border. Denoted as F_p
- A frame's **area** is the number of squares that are within the frame's border; note that this does *not* include the border squares themselves. Denoted as F_a

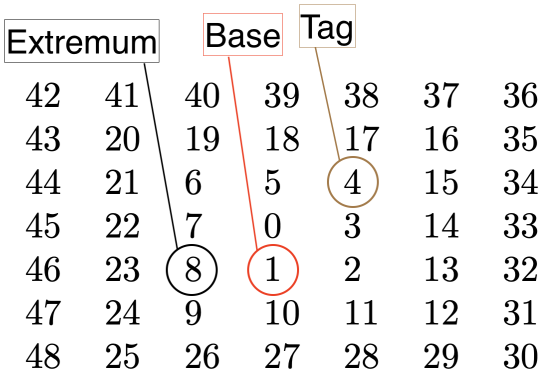

Figure 2: Extremum, Base, and Tag for F_1

Notice that all frames have 4 corners: for f_1, those corners are 6, 4, 2, and 8, in clockwise order. The total number of squares that make up the frame itself can then be expressed as 4 corners plus 4 times the number of squares per side.

To find the number of squares per side, notice how the board grows with each frame. Each additional frame adds 2 rows and 2 columns. Since we start with 1 row and 1 column, this implies that each frame has the i^{th} odd number of rows, where the 0^{th} odd number is 1. Thus, given a frame index f, the squares per side can be expressed as:

$$s = 2f - 1 \tag{2}$$

The frame's **perimeter** can then be written as:

$$F_p = 4(2f - 1) + 4 \tag{3}$$

Note that this could be rewritten as $F_p = 8f$; however, keeping the odd index key is useful, and provides symmetry for the area.

Since the frame's index can be directly linked to the length of it's side, the frame's area can be expressed simply as:

$$F_a = (2f - 1)^2 \tag{4}$$

Let us now inspect each frame's tag:

- f_0: $F_t = 0 = 0^2$
- f_1: $F_t = 4 = 2^2$
- f_2: $F_t = 16 = 4^2$
- f_3: $F_t = 36 = 6^2$

Notice a pattern — each frame's tag is a square of double the frame's index. To understand why, consider that each frame is a complete "square": the number of rows is the same as the number of columns. Because of this, by definition, the number of squares completely contained within a frame (i.e., the area **and** perimeter) will always be the square of a whole number. Where this index occurs is unimportant; its offset is completely determined by how you "start" the spiral (i.e., going down, to the left, etc.). In our implementation, we start by going down, so it happens that this squared value occurs in the top right corner. In general, as long as the indexing within each frame is continuous (which, in our case, it is), it can be shown that exactly one square along the border of the frame will contain the squared value. Thus, the tag can be written as:

$$F_t = (2f)^2 \tag{5}$$

Finding the indices of the four corners is now possible when we reference the frame's tag. Going counter-clockwise from the Frame's tag:

$$F_t = (2f)^2 = 4f^2$$
$$(2f)^2 + 2f = 2f(2f+1)$$
$$(2f)^2 + 4f = 4f(f+1)$$
$$(2f)^2 - 2f = 2f(2f-1)$$

In particular, note that the frame's extremum is found as:

$$F_e = 4f(f+1) \tag{6}$$

The frame's base value is a bit trickier. Suppose the extremum was actually the minimum: in other words, it was one less than the base value. Then, the base value would clearly just be the extremum plus 1. If we rewrite our extremum equation to reflect this supposition, we obtain:

$$(2f)^2 - 4f = 4f(f-1)$$

We can then add 1 to get the base value:

$$F_b = 4f(f-1) + 1 \tag{7}$$

3 Translation between Indices and Subscripts

There is a one to one relationship between indices and subscripts, since neither can be repeated. Further, a square has one, and only one, of each. In general, the two are not always mathematically related. Consider a board where indices are assigned randomly without replacement: though the one-to-one relationship still holds, there is clearly no mathematical relationship.

We used a square spiral so, in our case, there exists a mathematical relationship.

3.1 Conversion From a Subscript to an Index

Given the row and column subscripts, how do we determine where that square is within its frame? To answer this, let us notice that, for any frame F_f, there are 4 "stages" that its subscripts go through. We define the first stage as starting at the Frame's base value, F_b

- For the first stage, the row is held constant at $-f$, while the column subscript sweeps over $(-f, +f]$.
- For the second stage, the column is held constant at $+f$, while the row subscript sweeps over $[-f, +f]$.
- For the third stage, the row is held constant at $+f$, while the column subscript sweeps over $[+f, -f]$
- For the fourth and final stage, the column is held constant at $-f$, while the row sweeps over $[+f, -f]$

Thus, we can construct an equation, where r and c are from the subscript tuple, (r, c)

$$I = \begin{cases} F_e, & \text{iff } r = c = -f \\ F_b + 0f + f + c - 1, & \text{if } r = -f, c \neq -f \\ F_b + 2f + f + r - 1, & \text{if } c = +f \\ F_b + 4f + f - c - 1, & \text{if } r = +f \\ F_b + 6f + f - r - 1, & \text{if } c = -f, r \neq -f \end{cases}$$

Special Case for extremum Note the special case for the extremum; this is because that's where the frame's indices "wrap back around"

This can be simplified to

$$I = \begin{cases} F_e, & \text{iff } r = c = -f \\ F_b + 1f + c - 1, & \text{if } r = -f, c \neq -f \\ F_b + 3f + r - 1, & \text{if } c = +f \\ F_b + 5f - c - 1, & \text{if } r = +f \\ F_b + 7f - r - 1, & \text{if } c = -f, r \neq -f \end{cases}$$

Sloane Compatibility For compatibility, remove the negative offsets, and add a positive offset for the special case

$$I = \begin{cases} F_e + 1, & \text{if } r = -f, c = -f \\ F_b + 1f + c, & \text{if } r = -f, c \neq -f \\ F_b + 3f + r, & \text{if } c = +f \\ F_b + 5f - c, & \text{if } r = +f \\ F_b + 7f - r, & \text{if } c = -f, r \neq -f \end{cases}$$

The conversion from subscript to index can be completed as follows:

1. Identify the frame via equation 1
2. Find the frame's base value via equation 7
3. Using equation 3.1, we can find the index.

3.2 Conversion From an Index to a Subscript

For an index i:

$$f = \text{round}\left(\frac{\sqrt{i}}{2}\right) \tag{8}$$

Rounding Note that the rounding function must *always round up from* $n.5$. This is also known as "schoolbook rounding".

Then, to convert an index and frame back into its row and column, we use 3.1 in reverse:

$$r, c = \begin{cases} r = -f; c = (-f + (i - F_b) + 1, & \text{if } F_b \leq i < (F_b + 2f) \\ r = -f + (i - (F_b + 2f)) + 1; c = +f, & \text{if } (F_b + 2f) \leq i \leq F_t \\ r = +f; c = +f + (F_t - i), & \text{if } F_t < i \leq (F_t + 2f) \\ r = +f - (i - (F_t + 2f)); c = -f, & \text{if } (F_t + 2f) < i \leq F_e \end{cases}$$

Sloane Compatibility For compatibility, we must add back the offset; in this case, simply setting $i' = i - 1$, and then replacing i with i', will work. Note that this must be done in both equation 8 *and* equation 3.2

The steps can be written as:

1. Find the frame index with equation 8
2. Find the subscripts using equation 3.2

4 Moving the Knight

In principle, the knight's movement is relatively straightforward, even with the added restrictions in section 1

4.1 Possible Jump Destinations

We can express the possible moves for a given subscript as:

$$\begin{cases} r' = r \pm 1, r \pm 2 \\ c' = c \pm 2, c \pm 1 \end{cases}$$

These can combine into the following eight cases:

- $(r+1, c+2)$
- $(r+1, c-2)$
- $(r-1, c+2)$
- $(r-1, c-2)$
- $(r+2, c+1)$
- $(r+2, c-1)$
- $(r-2, c+1)$
- $(r-2, c-1)$

These subscripts can be converted into square indices using equation 3.1.

4.2 Removing Previously Visited Squares

The next step is to remove any positions the knight has already visited. Because the list of visited positions is potentially unbounded, membership testing needs to be as efficient as possible. To this end, using a Hash Set for the visited indices is suitable.

4.3 Picking the Lowest Index

The final step is to pick the smallest of the remaining indices. Once the minimum candidate is chosen, this index shall be added to the list of visited squares.

5 Example Implementation

Below we have given example implementations of the previous section information. It has been written in Swift, but the concepts discussed could be implemented in any language.

5.1 Data Types and Global Variables

Throughout the implementation, the following data types definitions will be useful:

```
typealias Frame = Int;
typealias Index = Int;

struct Subscript {
    let row: Int;
    let col: Int;
};

var indices: Set<Index> = [ 0 ];
// Maintain order (and compatibility) for comparison with Sloane Sequence
var sloanIndices: [Index] = [ 1 ];

var knight = Subscript(row: 0, col: 0);
```

5.2 Frame Functions

The following functions implement all of the functionality detailed in the framing section

```
func frameIndex(sub: Subscript) -> Frame {
    return max(abs(sub.row), abs(sub.col));
}

func frameIndex(idx: Index) -> Frame {
    return Frame(round(sqrt(Double(idx)) / 2));
}

func area(_ f: Frame) -> Int {
    return (2*f - 1) * (2*f - 1);
}

func perimeter(_ f: Frame) -> Int {
    return 8*f;
}

func extremum(_ f: Frame) -> Index {
    return 4*f*(f + 1);
}

func base(_ f: Frame) -> Index {
    return 4*f*(f - 1) + 1;
}
```

```
func tag(_ f: Frame) -> Index {
    return 4 * f * f;
}
```

5.3 Conversion Functions

These functions handle the conversion between subscripts and indices:

```
func sub2idx(sub: Subscript) -> Index {
    let f = frameIndex(sub: sub);
    let b = base(f);
    // special case!
    if sub.row == -f && sub.col == -f {
        return extremum(f);
    } else if sub.row == -f {
        return b + 1*f + sub.col - 1;
    } else if sub.col == f {
        return b + 3*f + sub.row - 1;
    } else if sub.row == f {
        return b + 5*f - sub.col - 1;
    } else if sub.col == -f {
        return b + 7*f - sub.row - 1;
    } else {
        return -1;
    }
}

func idx2sub(idx: Index) -> Subscript {
    let f = frameIndex(idx: idx);
    let b = base(f);
    let t = tag(f);
    let e = extremum(f);

    let r: Int;
    let c: Int;

    if b <= idx && idx < (b + 2*f) {
        r = -f;
        c = -f + (idx - b) + 1;
    } else if idx <= t {
        r = -f + (idx - (b + 2*f)) + 1;
        c = f;
    } else if idx <= (t + 2*f) {
        r = f;
        c = f + t - idx;
```

```
    } else if idx <= e {
        r = f - (idx - (t + 2*f));
        c = -f;
    } else {
        r = 0;
        c = 0;
    }

    return Subscript(row: r, col: c);
}
```

5.4 Destination Choosing

This function resolves the next position the knight should jump to. Note that if there are no possible options, then it will return nil:

```
func destination(_ knight: Subscript) -> Index? {
    return [
        Subscript(row: knight.row + 1, col: knight.col + 2),
        Subscript(row: knight.row + 1, col: knight.col - 2),
        Subscript(row: knight.row - 1, col: knight.col + 2),
        Subscript(row: knight.row - 1, col: knight.col - 2),
        Subscript(row: knight.row + 2, col: knight.col + 1),
        Subscript(row: knight.row + 2, col: knight.col - 1),
        Subscript(row: knight.row - 2, col: knight.col + 1),
        Subscript(row: knight.row - 2, col: knight.col - 1),
        ].map(frameIndex(sub:))
        .filter(indices.contains)
        .min();
}
```

5.5 Putting it All Together

The implementation is relatively straightforward:

```
while let jump = destination(knight) {
    indices.insert(jump);
    sloanIndices.append(jump + 1);
    knight = idx2sub(idx: jump);
}
```

6 Future Work

While this is the complete implementation for the base problem, as given by Neil Sloane, there are variations. In this section, we shall explore these possibilities.

6.1 Different Pieces

Future work could focus on movement types other than the knight's: a bishop, for example. Customized movements, not seen in chess, could also be explored.

Furthermore, as is the case for the bishop, the number of possible destinations is infinite — this implies that sections 4.1, 4.2, and 4.3 will overlap more than they do now.

6.2 Different Indexing Mechanisms

The Trapped Knight problem uses the square spiral to index the squares. As alluded to earlier, this is not the only way. For instance, Neil Sloane introduces a *diagonal* method as well: the piece starts in the top left of the board, and indexing goes "up the diagonals". This is perhaps easiest to explain with an example; see table 4. Notice that the top left is 0; every diagonal "starts" at its leftmost square, and "ends" at its rightmost square.

0	2	5	9	14	20	27
1	4	8	13	19	26	33
3	7	12	18	25	32	38
6	11	17	24	31	37	42
10	16	23	30	36	41	45
15	22	29	35	40	44	47
21	28	34	39	43	46	48

Table 4: Diagonally-indexed chess board

6.3 Different Selection Criterion

In the problem as given, the square with the smallest *absolute* index is chosen. However, it may be interesting to instead pick the "closest" square instead: the square whose index is the shortest distance away from the current index.

A slightly more complicated set of criteria might attempt to select the closest square whose index is less than our own; if no such square exists, then pick the closest square. For example, if the current index is 7:

- For choices [1, 8, 9], we would choose 1, because it is the only choice less than the current index.
- For choices [1, 6, 8, 9], we would choose 6, because 6 is closer to 7 than 1 is, while still being less than 7.
- For choices [8, 9, 10], we would choose 8, because it is the closest value to 7, and there are no choices less than 7. Note that this would always be identical to the base problem specification.

This would require further investigation in section 4.3, since we are no longer blindly picking the lowest index.

A Unit Tests

To test the basic functionality, the following code is a rudimentary unit testing framework for the Swift code.

Note that it is the bare minimum. To ensure complete interoperability, more tests should be added.

```
protocol UnitTest {
    associatedtype I;
    associatedtype O;

    var input: I { get };
    var output: O { get };
    func test() -> Bool;
}
struct FrameIndexSubscriptTest: UnitTest {
    let input: Subscript;
    let output: Frame;
    func test() -> Bool {
        let out = frameIndex(sub: input);
        if out != output {
            print("Expected: \(output); Got \(out) instead");
        }
        return out == output;
    }
}
struct FrameIndexIndexTest: UnitTest {
    let input: Index;
    let output: Frame;
    func test() -> Bool {
        let out = frameIndex(idx: input);
        if out != output {
            print("Expected: \(output); Got \(out) instead");
        }
        return out == output;
    }
}
func testFrameIndex() -> Bool {
    let subTests: [FrameIndexSubscriptTest] = [
        FrameIndexSubscriptTest(input: Subscript(row: 0, col: 0), output: 0),
        FrameIndexSubscriptTest(input: Subscript(row: 1, col: 1), output: 1),
        FrameIndexSubscriptTest(input: Subscript(row: 0, col: -2), output: 2),
        FrameIndexSubscriptTest(input: Subscript(row: 7, col: -7), output: 7),
        FrameIndexSubscriptTest(input: Subscript(row: 10, col: -15), output: 15),
    ];
```

```
    let idxTests: [FrameIndexIndexTest] = [
        FrameIndexIndexTest(input: 0, output: 0),
        FrameIndexIndexTest(input: 8, output: 1),
        FrameIndexIndexTest(input: 1, output: 1),
        FrameIndexIndexTest(input: 9, output: 2),
        FrameIndexIndexTest(input: 48, output: 3),
    ];

    for t in subTests {
        if !t.test() {
            return false;
        }
    }

    for t in idxTests {
        if !t.test() {
            return false;
        }
    }

    return true;
}
struct Sub2IdxTest: UnitTest {
    let input: Subscript;
    let output: Index;
    func test() -> Bool {
        let out = sub2idx(sub: input);
        if out != output {
            print("Expected: \(output); Got \(out) instead");
        }
        return out == output;
    }
}
func testSub2Idx() -> Bool {
    let tests: [Sub2IdxTest] = [
        Sub2IdxTest(input: Subscript(row: 0, col: 0), output: 0),
        Sub2IdxTest(input: Subscript(row: -1, col: 0), output: 1),
        Sub2IdxTest(input: Subscript(row: -1, col: 1), output: 2),
        Sub2IdxTest(input: Subscript(row: 0, col: 1), output: 3),
        Sub2IdxTest(input: Subscript(row: 1, col: 1), output: 4),
        Sub2IdxTest(input: Subscript(row: 1, col: 0), output: 5),
        Sub2IdxTest(input: Subscript(row: 1, col: -1), output: 6),
        Sub2IdxTest(input: Subscript(row: 0, col: -1), output: 7),
        Sub2IdxTest(input: Subscript(row: -1, col: -1), output: 8),
        Sub2IdxTest(input: Subscript(row: -2, col: -1), output: 9),
        Sub2IdxTest(input: Subscript(row: -2, col: 0), output: 10),
```

```
        Sub2IdxTest(input: Subscript(row: -2, col: 1), output: 11),
        Sub2IdxTest(input: Subscript(row: -2, col: 2), output: 12),
        Sub2IdxTest(input: Subscript(row: -1, col: 2), output: 13),
        Sub2IdxTest(input: Subscript(row: 0, col: 2), output: 14),
        Sub2IdxTest(input: Subscript(row: 1, col: 2), output: 15),
        Sub2IdxTest(input: Subscript(row: 2, col: 2), output: 16),
        Sub2IdxTest(input: Subscript(row: 2, col: 1), output: 17),
        Sub2IdxTest(input: Subscript(row: 2, col: 0), output: 18),
        Sub2IdxTest(input: Subscript(row: 2, col: -1), output: 19),
        Sub2IdxTest(input: Subscript(row: 2, col: -2), output: 20),
        Sub2IdxTest(input: Subscript(row: 1, col: -2), output: 21),
        Sub2IdxTest(input: Subscript(row: 0, col: -2), output: 22),
        Sub2IdxTest(input: Subscript(row: -1, col: -2), output: 23),
        Sub2IdxTest(input: Subscript(row: -2, col: -2), output: 24),
        Sub2IdxTest(input: Subscript(row: -3, col: -2), output: 25),
    ];

    for t in tests {
        if !t.test() {
            print(t);
            return false;
        }
    }
    return true;
}
testFrameIndex();
testSub2Idx();
```