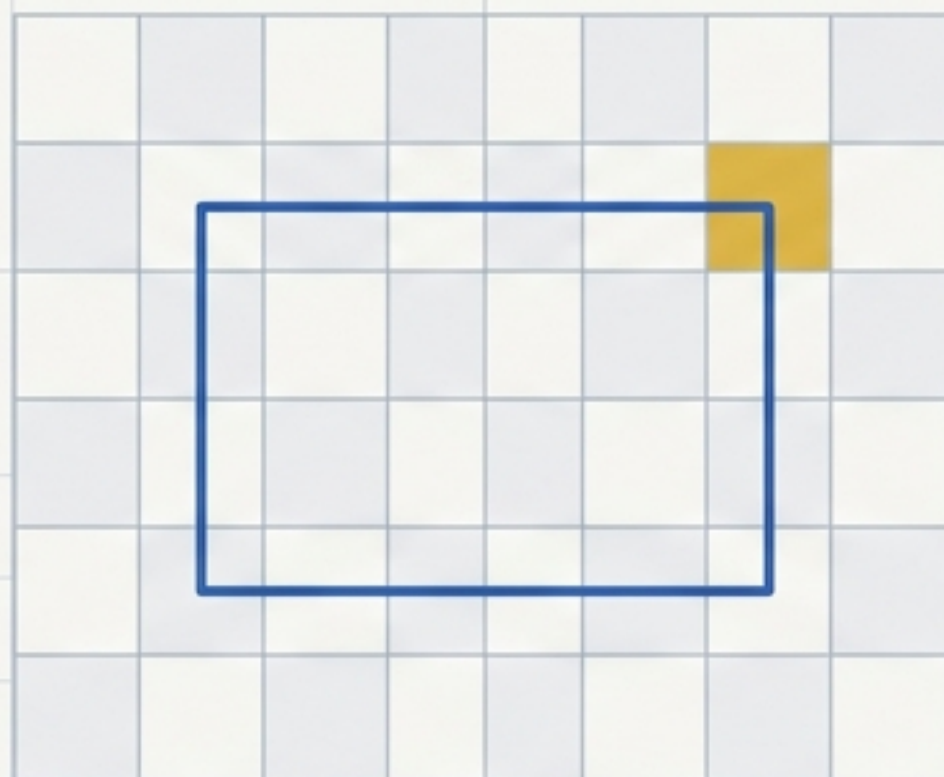


Pythonによるナイトの巡歴問題の実装と最適化

深さ優先探索の解剖と「番兵」によるパラダイムシフト



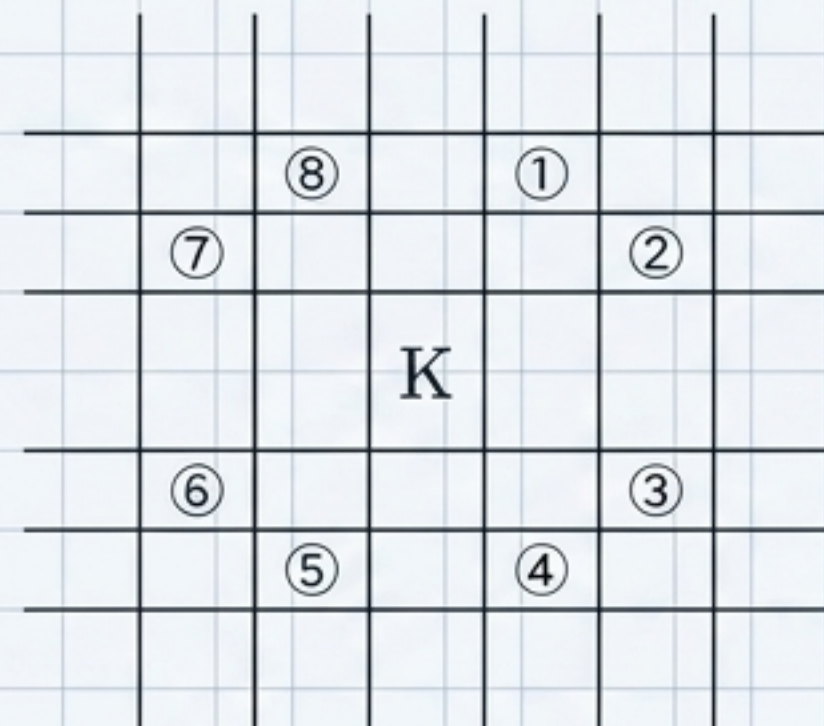
応用情報技術者試験（平成30年春期 午後問3）
解析ドキュメント

The Challenge: ナイトの巡歴問題とは？

目的: $M \times N$ マスの盤面上をナイトが移動し、全てのマスを1回ずつ通過する経路を発見する。

ルール: チェスの「ナイト」の動き（L字型）。

アルゴリズム: 再帰関数を用いた「深さ優先探索（DFS）」で全探索を実行する。



		列		
		1	2	3
行	1	1	12	3
	2	4	9	6
	3	7	2	11
	4	10	5	8

注記 マス内の数はナイトが移動する順序を表す。

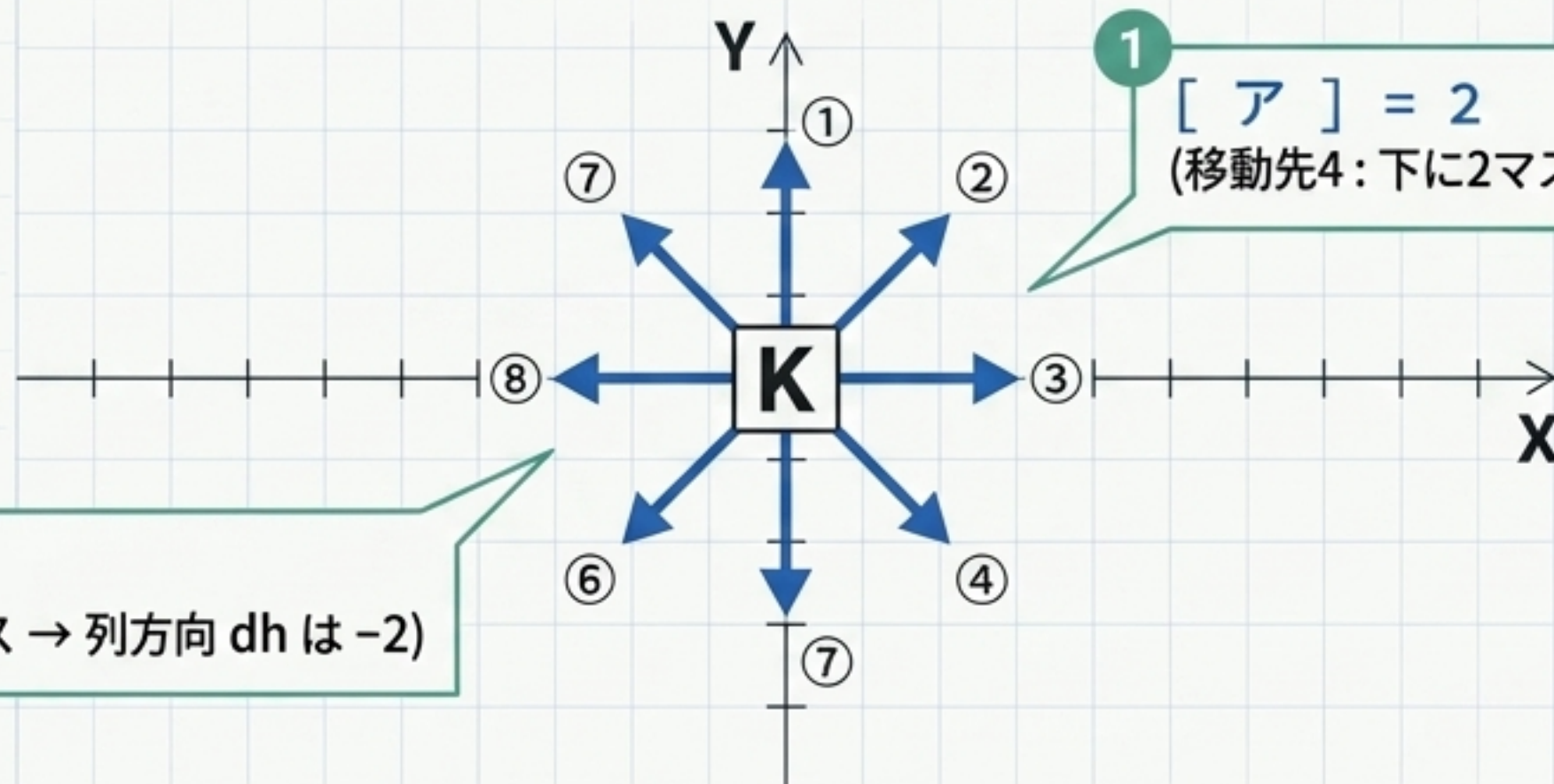
図2 4×3 マスの場合の解の一つ

Movement as Data: 配列による移動ベクトル

8方向の物理的な移動を、行方向 dv と列方向 dh の変化量（ベクトル）として配列に格納する。

表1 $dv[]$, $dh[]$ に設定する値

図1の移動先	①	②	③	④	⑤	⑥	⑦	⑧
$dv[]$	-2	-1	1	ア	2	1	-1	-2
$dh[]$	1	2	2	1	-1	イ	-2	-1



1
[ア] = 2
(移動先4: 下に2マス、右に1マス → 行方向 dv は +2)

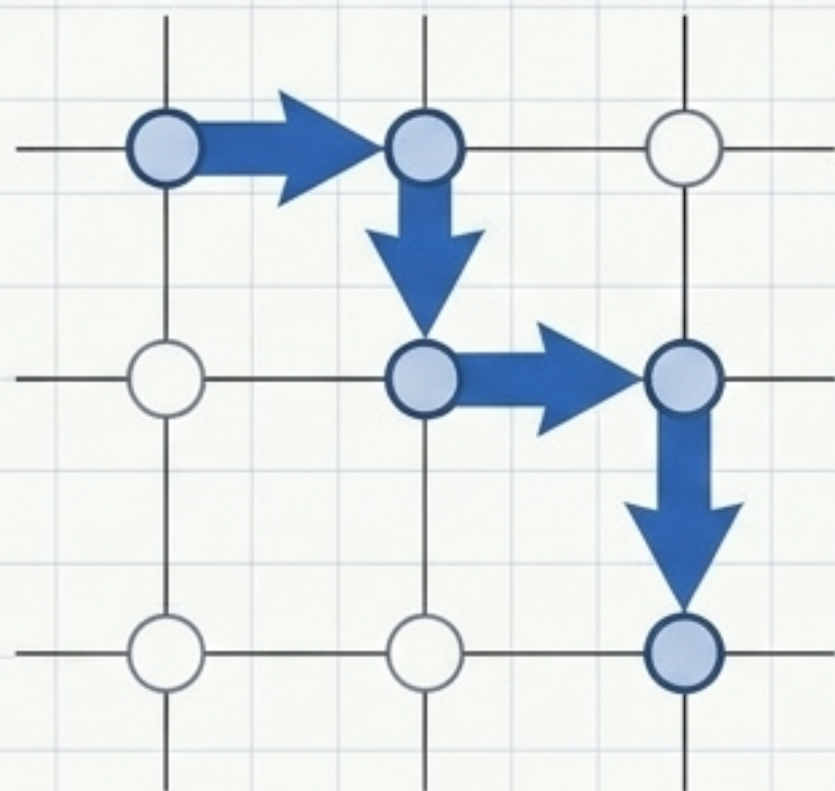
2
[イ] = -2
(移動先6: 下に1マス、左に2マス → 列方向 dh は -2)

Core Engine: 深さ優先探索とバックトラック

探索と取り消し（行きがけと帰りがけ）の往復運動が全探索を可能にする。

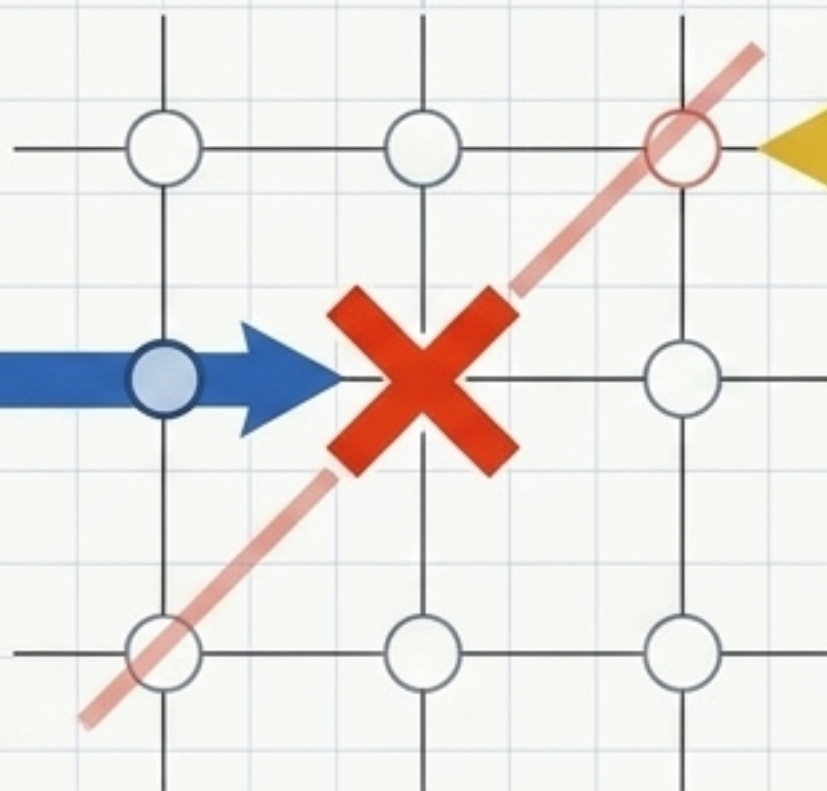
1. Go Deep

進める限り深く探索する



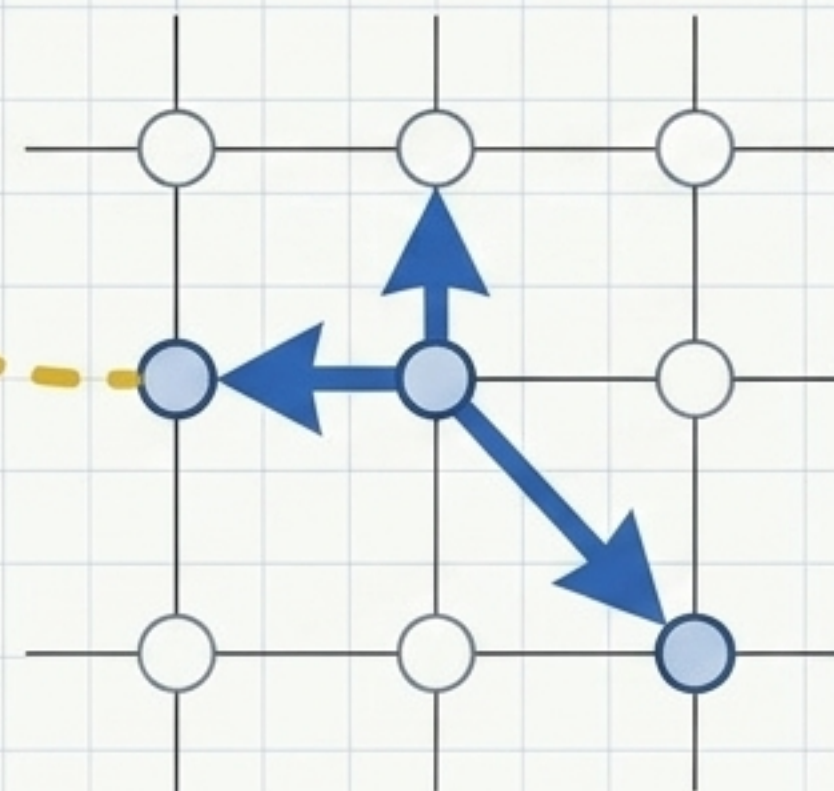
2. Hit Wall

行き止まり（範囲外・訪問済み）に到達



3. Step Back

1歩戻り（バックトラック）、別の分岐を試す



The Python Implementation: search() 関数の全容

4つの論理ブロックが組み合わさり、自己相似的（再帰的）に経路を開拓する。

```
1: def search(i, v, h):
2:     if (1 <= v <= m) and (1 <= h <= n) and (board[v][h] == 0): # 境界判定と未訪問判定
3:
4:
5:         board[v][h] = i # 移動順序を記録
6:         if i == m * n: # ゴール判定
7:             printBoard() # 解答を印字
8:             printFlag = 1 # 解答発見フラグを設定
9:     else:
10:         for j in range(8): # 次の移動先を調べる
11:             nv, nh = v + dv[j], h + dh[j]
12:             search(i + 1, nv, nh) # 再帰呼び出し
13:
14:         board[v][h] = 0 # バックトラック（移動順序を取り消す）
15:
16:
17:
18:
```

境界判定
(Boundary Check)

記録とゴール判定
(Validation & Goal)

再帰呼び出し
(Recursive Branching)

バックトラック
(Backtracking)

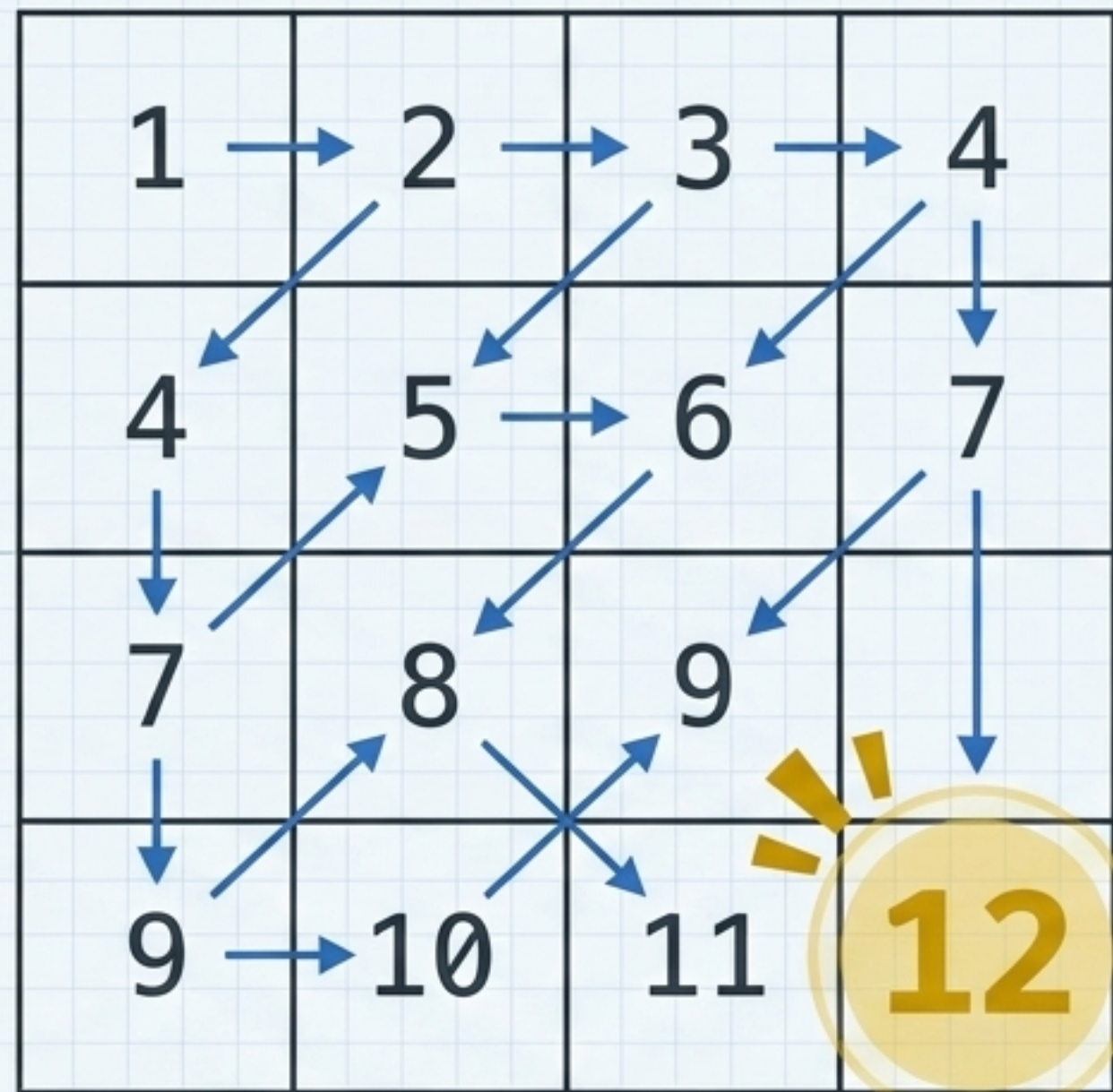
Block A: 記録とゴール判定

```
4: board[v][h] = i
5: if i == m * n:
6:     printBoard()
7:     printFlag = 1
8: else:
```

`board[v][h] = i`: 現在の手数 (移動順序 i) を盤面に記録。

`[ウ] = i == m * n`

意味: 現在の手数 i が、盤面の総マス数 $m * n$ と等しければ、すべてのマスを巡歴した (クリア) と判定。



巡歴完了: 最後のマスに手数 '12' を記録。

Block B: 次のマスへの再帰分岐

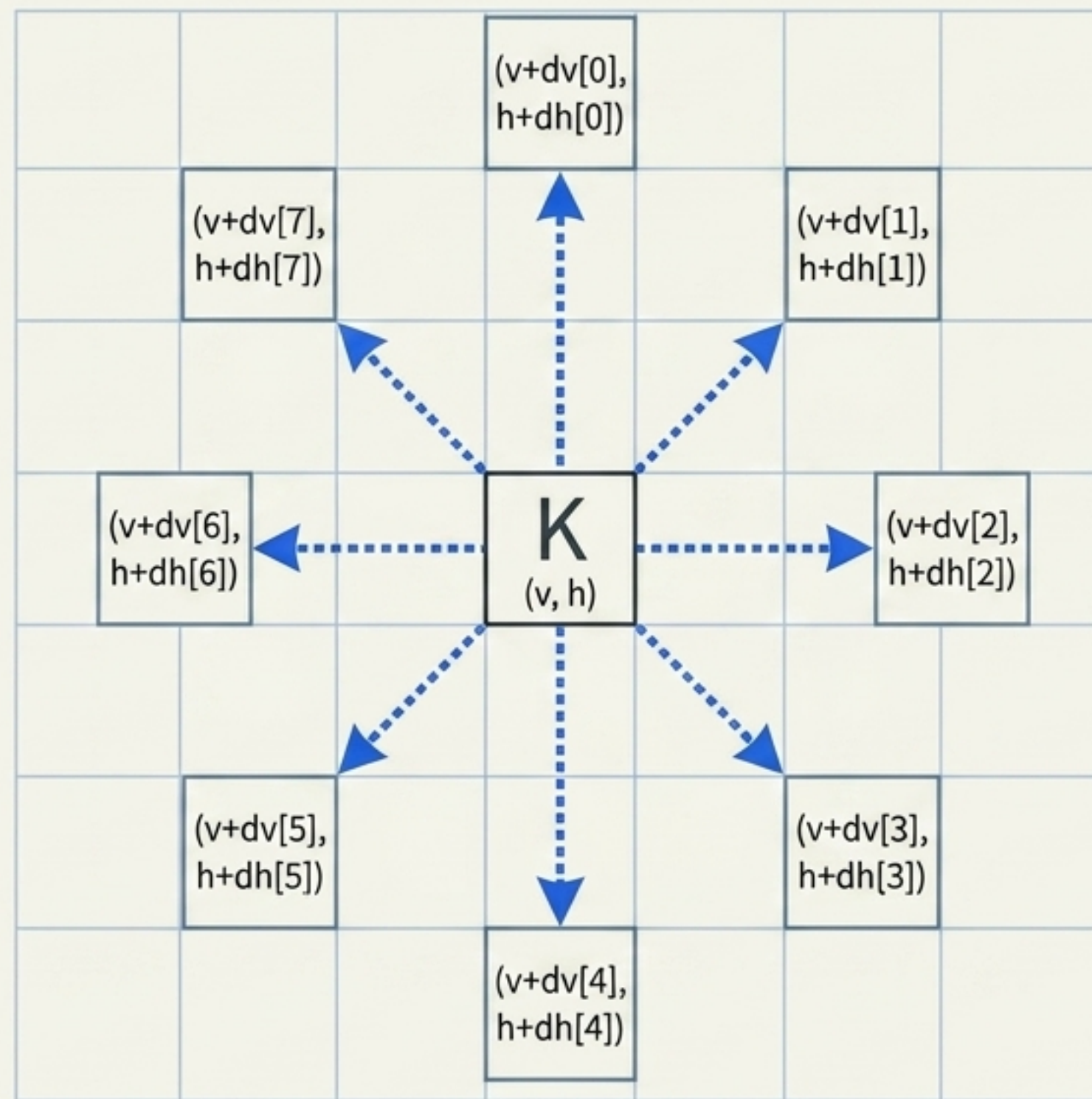
```
10:     for j in range(8):
11:         nv, nh = v + dv[j], h + dh[j]
12:         search(i + 1, nv, nh)
```

for文で1～8方向を順に調べ、自分自身
(search関数) を呼び出す。

[エ] 次の手数: $i + 1$

[オ] 次の行座標: $v + dv[j]$

[カ] 次の列座標: $h + dh[j]$



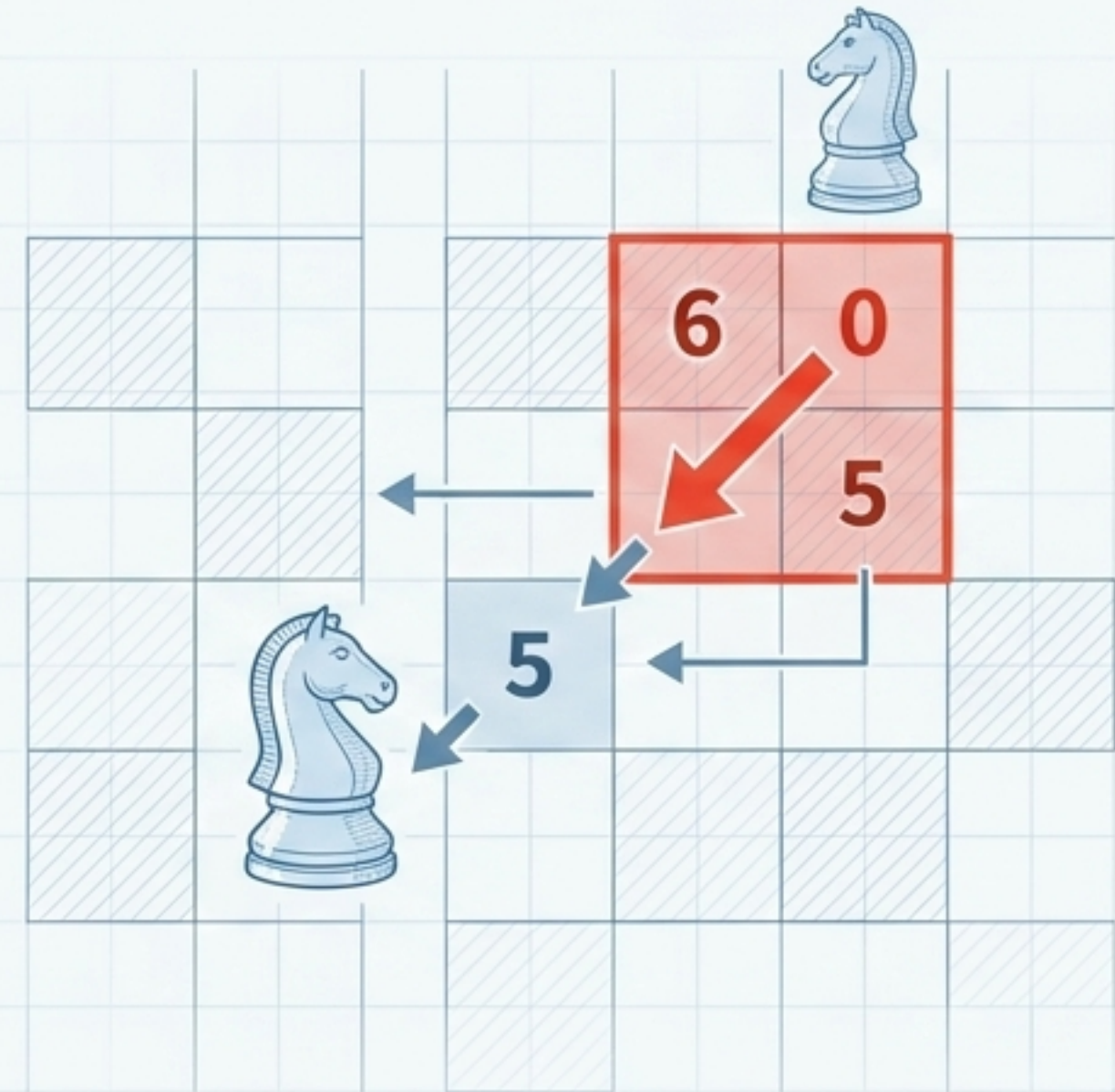
Block C: 探索の取り消し（バックトラック）

```
13:   endif
14:   [ キ ] # board[v][h] = 0
15:   endif
```

8方向すべて試しても解がない場合、そのルートは「ハズレ」。元のマスに戻る前に、自分の足跡を消す必要がある。

[キ] = board[v][h] = 0

意味: 現在のマスを「未訪問（初期値）」にリセットし、別の道を試せるようにする。



バックトラック: 現在のマスの値 ('6') を '0' にリセットし、前のマス ('5') に戻る。

The Hidden Friction: 境界チェックの代償

```
1: function search( i, v, h )
2:   if( v >= 1 and v <= m )
3:     if( h >= 1 and h <= n )
4:       if( board[v][h] == 0 )
5:         board[v][h] ← i
6:         if( ウ )
7:           printBoard()
8:           printFlag ← 1
9:         else
10:          for( j を 1 から 8 まで 1 ずつ増やす )
11:            search( エ , オ , カ )
12:          endfor
13:        endif
14:      キ
15:    endif
16:  endif
17: endif
18: endfunction
```



Checks performed:
1,000,000+

Insight: ナイトが移動するたびに、「盤面の外にはみ出していないか？」を行と列で毎回確認している。探索が深くなるほど、この if 文は数百万回実行され、アルゴリズムの大きな無駄（ボトルネック）となる。

The Paradigm Shift: 「番兵 (Sentinel)」の導入

Concept: 盤面の外側に「見えない壁 (値が1のマス)」を敷き詰めるという発想の転換。

Execution: 元の盤面の上下左右に2マス分の余白を追加し、初期値を1 (範囲外) で埋める。
本来の探索範囲のみ0 (未訪問) とする。

変更前の盤面

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

変更後の盤面

	1	2	3	4	5	6	6	7
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	0	0	0	0	1	1
4	1	1	0	0	0	0	1	1
5	1	1	0	0	0	0	1	1
6	1	1	0	0	0	0	1	1
7	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1

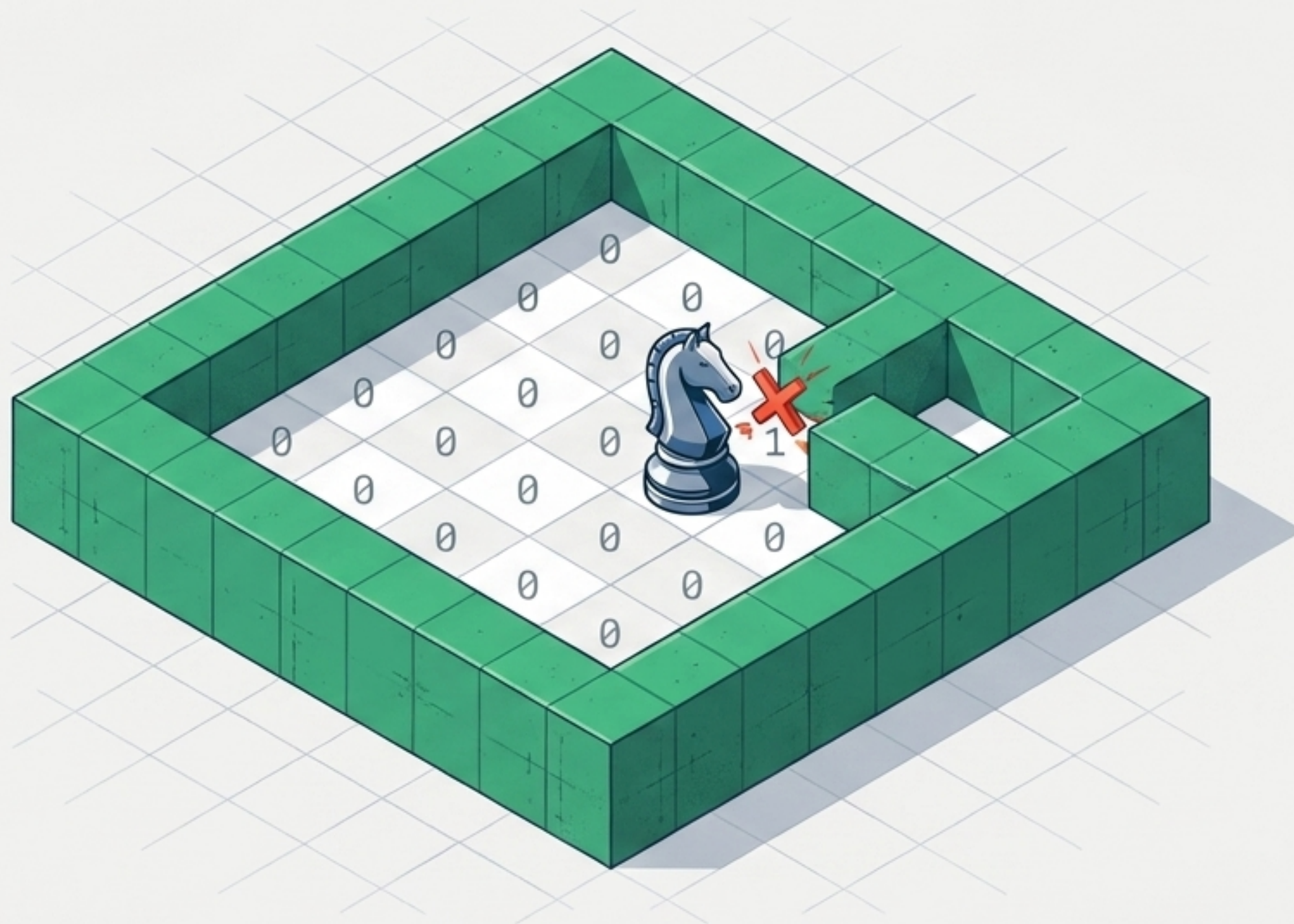
The "Moat" Metaphor: 壁が境界判定を代行する

1. プログラムの4行目

`if board[v][h] == 0` (未訪問チェック)
がある。

2. 番兵 (1) を配置したことで、盤面外に飛び出してもこの条件で自動的にブロックされる。

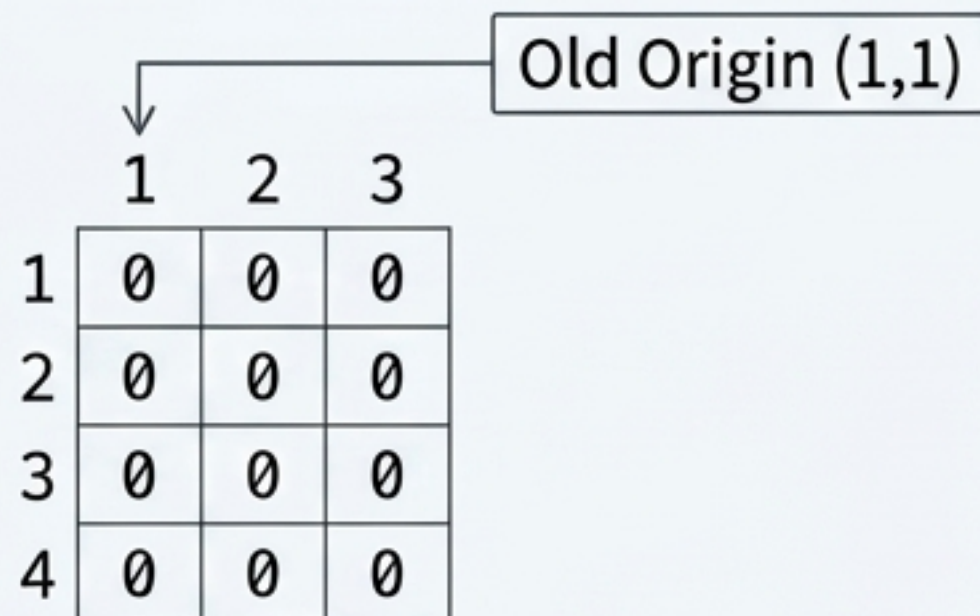
3. 結果: 手動での境界判定が完全に不要になる！



Coordinate Shift: 座標系のアップデート

盤面全体が右下にシフトしたため、開始位置と出力範囲を調整する。

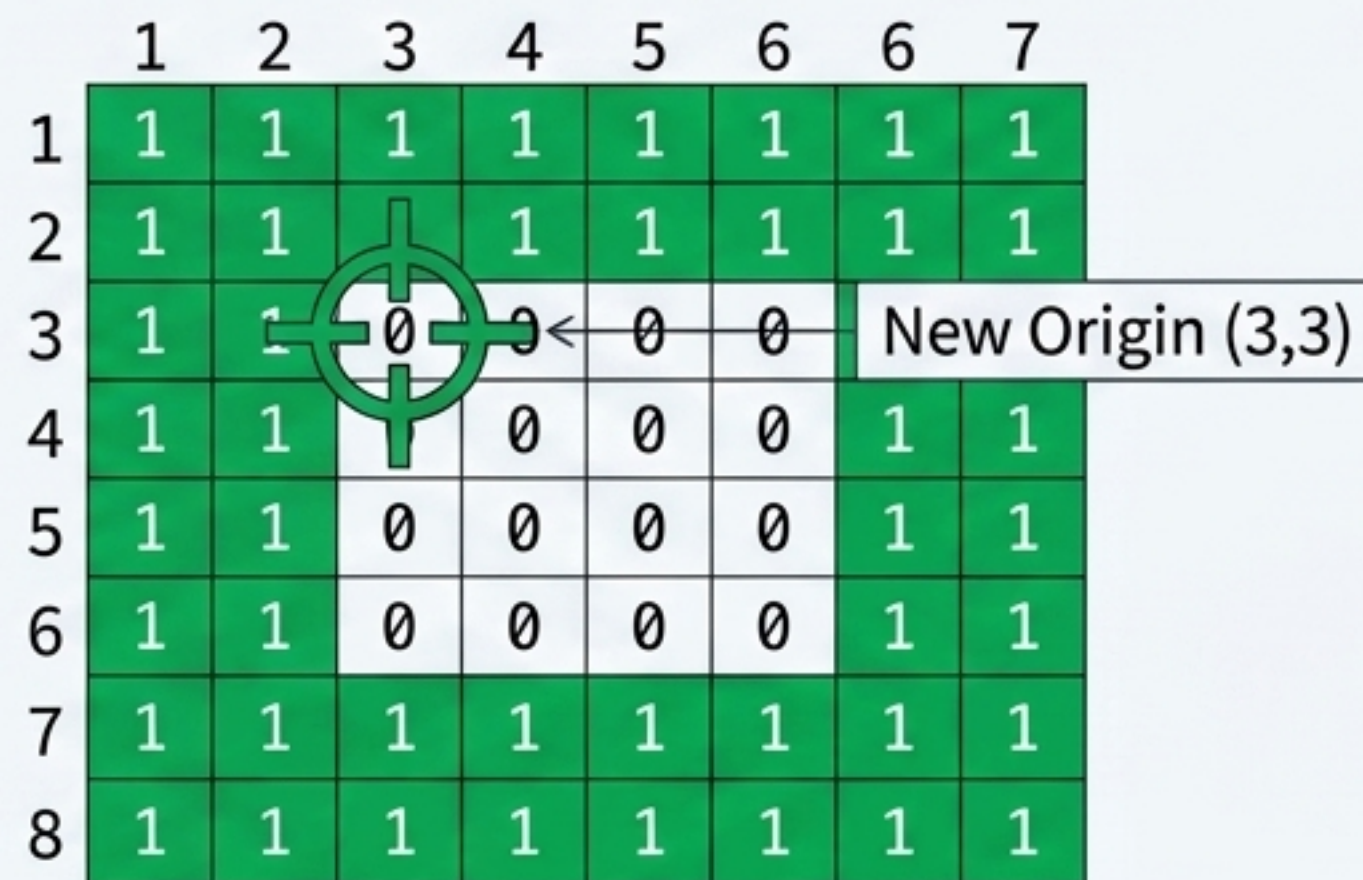
Old Coordinate System



Old Origin (1,1)

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

New Sentinel Coordinate System



	1	2	3	4	5	6	6	7
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	0	0	0	0	1	1
4	1	1	0	0	0	0	1	1
5	1	1	0	0	0	0	1	1
6	1	1	0	0	0	0	1	1
7	1	1	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1

New Origin (3,3)

Update 1: 印字範囲

printBoard() のループを `for v in range(3, m+3)` と `range(3, n+3)` に変更。

Update 2: スタート地点

メインプログラムの呼び出しを `search(1, 3, 3)` に変更。

The Payoff: 不要コードのパーズ

番兵の導入により、複雑だった多重 if 文が消滅。2, 3, 16, 17行目を丸ごと削除。

Before: Original Code with Boundary Checks

```
1 def search(v, h):
2     if (v >= 1 and v <= n) and (h >= 1 and h <= m): # Boundary Check
3     if board[v][h] == 0: # Unvisited Check
4         board[v][h] = 1 # Mark visited
5         print(f'Visiting: ({v}, {h}))')
6         # Explore neighbors (U, D, L, R)
7         search(v-1, h)
8         search(v+1, h)
9         search(v, h-1)
10         search(v, h+1)
11         # Backtrack
12         board[v][h] = 0
13         print(f'Backtracking from: ({v}, {h}))')
14     else:
15         print(f'Path Blocked or Visited: ({v}, {h}))')
16     else:
17         print(f'Out of Bounds: ({v}, {h}))')
```

After: Refactored Code with Sentinel

```
1 def search(v, h):
2 +     if board[v][h] == 0: # Unvisited and not a wall (sentinel)
3 +         board[v][h] = 1 # Mark visited
4 +         print(f'Visiting: ({v}, {h}))')
5 +         # Explore neighbors (U, D, L, R)
6 +         search(v-1, h)
7 +         search(v+1, h)
8 +         search(v, h-1)
9 +         search(v, h+1)
10 +        # Backtrack
11 +        board[v][h] = 0
12 +        print(f'Backtracking from: ({v}, {h}))')
```

Result: コードが圧倒的に短く、読みやすく、そして高速に。

Synthesis: アプローチの比較表

	標準アプローチ (Standard)	番兵アプローチ (Sentinel)
境界判定	毎回のif文で手動チェック	配列のデータ自体が壁として機能
コード量	多重ネストで冗長	if文が消え、極めてシンプル
メモリ空間	最小限 ($M \times N$)	やや増加 (余白分拡大)
実行速度	条件分岐による摩擦	条件分岐カットによる高速化



データ構造の設計が、 アルゴリズムの美しさを決める。

複雑な if 文（ロジック）で問題をねじ伏せるのではなく、データ構造（配列の形状）を工夫することで、プログラムは自然とシンプルになる。「番兵」はその最高のエレガンスの実例である。