課題

地引　昌弘

2021.12.09

課題 1 — ナイトの到達範囲

古代インドで発明された**チャトランガ** (Chaturanga) はその後世界中に広まり、日本では将棋、欧米圏では**チェス** (Chess) と呼ばれ、ルールに多少の違いはあるものの極めて一般的なボード ゲームの一つとして現代に伝わっています。チェスは、8 行 × 8 列の計 64 マスからなる盤上に 6 種類の駒を並べ、自分と相手プレーヤとで交互に一駒ずつ動かして行きます。駒は、その種類に応じて動けるマスが決まっており、**キング** (King) と呼ばれる駒を取った方が (相手キングが存在するマスに自分の駒のどれかが到達した方が) 勝ちとなります[1]。ところで、チェスで使われる駒の一つに、**ナイト** (Knight) と呼ばれる駒があります。この駒は騎士を表わしていると言われ、他の駒を飛び越えられる変わった動きをします (図 1)。ナイトの駒が図 1 のマス S にある場合、このナイトは 1 回の移動で赤丸マスのどれか一つに動くことができます。

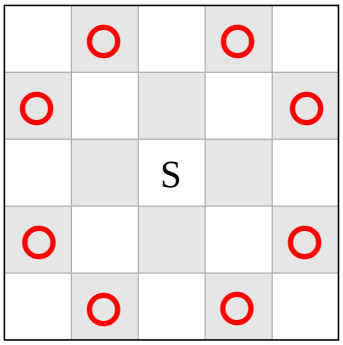

図 1: ナイトが 1 回の移動で到達できるマス

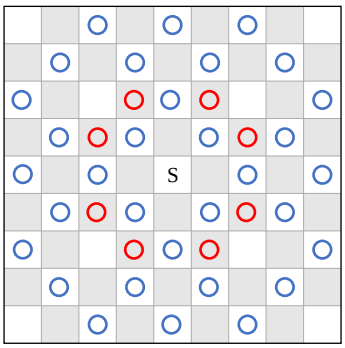

図 2: ナイトが 2 回の移動で到達できるマス

ナイトの動きはこのように独特であり、白黒の市松模様であるチェス盤との関係から、数理科学の世界において、これまで様々なモチベーションを与えて来ました。以下では、チェスの勝負から少し離れ、このナイトの動きに纏わるアルゴリズムを考えてみましょう。

図 2 は、ナイトを 2 回動かした時に到達できるマスを示しています。S のマスにあるナイトは、1 回目で図 2 にある赤丸マス (のどれか) へ移動し、これらの各マスから 2 回目の移動で青丸マス (のどれか) へ到達できます。チェスの盤は白と黒の市松模様になっており、図 1 と図 2 を見比べてみると、各回では元の色と違う色のマスへ移動していることが分かります。

次に、ナイトを 3 回動かして到達できるマスを図 3 (2 ページ) に示します。3 回目の移動では、到達できるマスが増えて見えにくいので、図 3 は、2 回目までに到達したマスを薄灰丸で、3 回目で到達したマスを濃灰丸で表わしました。これを見ると、ある一定領域内 (図 3 の黒実線で示した八角形) にある黒マスは、全て到達可能であることが分かります。

最後に、ナイトを 4 回動かして到達できるマスを図 4 (2 ページ) に示します。さすがに 4 回目の移動となると、到達できるマスが大幅に増えて紙面に入り切れないため、図 4 はマス S を原点とした第一象限だけを取り出しました。図 4 では、3 回目までに到達したマスを薄/濃灰丸で、4 回目で到達したマスを薄緑丸で表わしています。これを見ると、3 回目の移動と同様に、ある一定領域内 (図 4 の黒点線で示した八角形) にある白マスは、全て到達可能であることが分かります。

[1] とは言え、将棋やチェスでは、(一度に動けるマスは決まっていますが) 王将 or キングの駒は盤上のどこにでも自由に動けるので、相手の駒が迫って来ると当然逃げます。つまり、そう簡単には相手キングのマスに自分の駒を到達させることはできません。参考ですが、中国にもチャトランガを由来とする**シャンチー** (象棋) と呼ばれるゲームがあります。シャンチーでは、盤の中央に盤を二分する河と呼ばれるマス (と言うかマス行) があり (河を渡れる駒は制限される)、また、王将 or キングに該当する駒 (帥将と呼ばれる) が初期位置からあまり動けない (つまり、皇帝は王宮から外に出られない) といった、中国の歴史を反映したルールになっています。

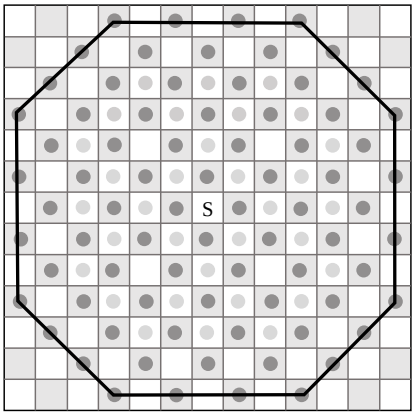

図 3: ナイトが 3 回の移動で到達できるマス

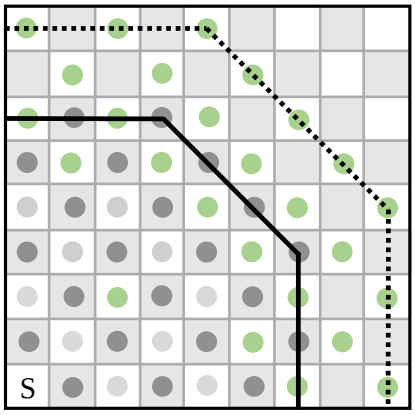

図 4: ナイトが 4 回の移動で到達できるマス

さて、以上の実験で見えて来た八角形の領域を移動回数 n と紐付け、以下では “**第 n 八角形**” と呼ぶことにしましょう。第 n 八角形は、下記の特徴を持っています。

A: 第 $n+1$ 八角形は、第 n 八角形より、1) 縦横方向は 1 列 or 1 行離れている。2) 斜め方向は 2 斜め並び離れている[2]。

B: 始点 S を白マスとした場合、$n \geq 3$ の n に対して、n が奇数の時は第 n 八角形内の全黒マスが、n が偶数の時は同じく全白マスが、S にあるナイトから n 回の移動で到達可能である[3]。

特徴 B が成り立つことは、簡単に証明できます。

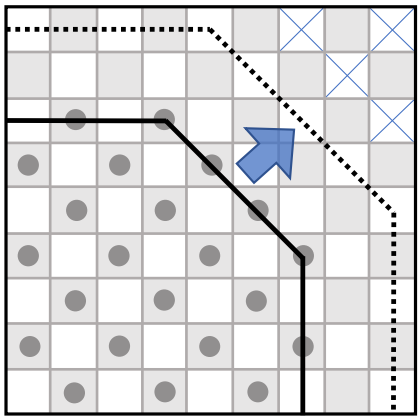

図 5: 第 n 八角形から第 $n+1$ 八角形への遷移

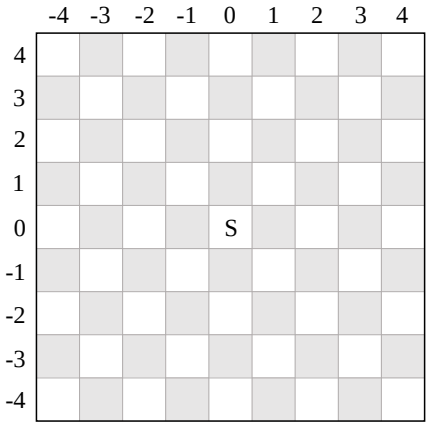

図 6: 盤の座標系

略証:

図 1 (1 ページ) で示したように、ナイトの移動規則では、移動先は元のマスから 2 マス離れた異なる色のマスになる。以下、図 5 を例に、第 n 八角形 (黒実線) から第 $n+1$ 八角形 (黒点線) へ遷移する場合を考える。ここで、ナイトは第 n 八角形内の全黒マスに到達するが、第 $n+1$ 八角形内には到達できない白マス a が存在すると仮定する。

i) a が第 n 八角形に含まれる場合:
$n \geq 3$ の第 n 八角形内には、a から 2 マス離れた黒マス b が必ず存在する。ナイトは第 n 八角形内の全黒マスに到達することから、b に到達できる。この時、$n \to n+1$ の遷移において a に到達できないことから、ナイトは b から a に到達できない。これは、ナイトの移動規則と矛盾する。

ii) a が第 n 八角形に含まれず、第 $n+1$ 八角形に含まれる場合:
特徴 A より、a から 2 マス離れた黒マス b が、$n \geq 3$ の第 n 八角形内に必ず存在する。以下、i) と同じ議論により矛盾を導ける。

[2]図 4 では、黒斜め点線は同実線より、1. 薄緑丸の斜め並び, 2. 黒マスの斜め並びの計 2 斜め並びだけ離れています。
[3]始点 S が黒マスの場合は、上記の白と黒を入れ替えた同じ特徴を持ちます。

どちらの場合も矛盾が生じるため、第 $n+1$ 八角形内にはナイトが到達できない白マス a は存在しない。よって、$n \rightarrow n+1$ の遷移では、図 5 の右上にある×を付けた白マス以外に全て到達できる。

以上で、ナイトの動作に伴う特徴的なパターンの紹介を終え、以下では、これに関連する演習問題に取り組むことにしましょう。まずは、図 6 (2 ページ) のように、ナイトの初期位置を (0, 0) とする座標系を定義します[4]。続いて、この座標系に関連する下記の関数 init_board() と set_mark() を用意します。

```
import ita

def init_board(n):
  size_i = size_j = 4*n + 1 + 2    # "+ 2" は、盤の上下左右端に 1 行ずつ空行を入れるため。
  b = ita.array.make2d(size_i, size_j)
  for i in range(size_i):
    for j in range(size_j):
      b[i][j] = 0
  b[2*n + 1][2*n + 1] = 5          # 始点 S（座標系の原点）と盤配列 b の添字番号との関係に注意
  return(b)

def set_mark(b, x, y, v):
  o = int((len(b) - 1)/2)          # o = 始点 S を表わす盤配列 b の添字番号
                                   # （対応関係は init_board() 内のコメントを参照）
  j = o + x
  i = o - y                        # 座標系の (x, y) と配列の i 行・j 列との対応に注意
  b[i][j] = v
  return
```

init_board() は、ナイトの移動回数 n を引数として渡すと、n に応じた適当なサイズの盤を配列 b として返します。配列 b 内では、ナイトの初期位置 (始点 S/座標系の原点) を数値の 5 で表わし (対応する要素に 5 が格納され)、他のマスは 0 で表わします。動作イメージは下記の通り。

```
>>> b = init_board(1)          ← ナイトの 1 回移動に対応する盤を作成します。
>>> import pprint
>>> pprint.pprint(b, width=100)
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 5, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0]]
>>>
```

set_mark() は、それぞれ引数として渡した盤配列 b の座標 (x, y) に数値 v を設定します。動作イメージは下記の通り。

```
>>> set_mark(b, 1, 2, 1)       ← 盤の座標 (1, 2) へ数値 1 を設定します。
>>> pprint.pprint(b, width=100)
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 5, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0]]
>>>
```

[4] この例では、(-4, -4) ～ (4, 4) までしか記載していませんが、移動回数の増加により盤が大きくなった場合は、当然それに合わせて (-i, -i) ～ (i, i) まで拡張されるので、勘違いしないように。

課題 1-a: 関数 init_board() および set_mark() を利用し、次の仕様を満たす関数 get_octagon() を作成せよ。

1. 移動回数 n を指定すると、第 n 八角形に該当するマスを数値 1 で表わした盤配列 b を返す。

2. 但し、返値となる盤配列 b では、ナイトの始点 S に該当するマスは数値 5 で表わす。

get_octagon() の動作イメージは下記の通り。

```
>>> b = get_octagon(1)
>>> pprint.pprint(b, width=100)
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 1, 1, 0, 0],
 [0, 1, 1, 1, 1, 1, 0],
 [0, 1, 1, 5, 1, 1, 0],
 [0, 1, 1, 1, 1, 1, 0],
 [0, 0, 1, 1, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 0]]
>>> b = get_octagon(2)
>>> pprint.pprint(b, width=100)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
 [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 1, 1, 1, 1, 5, 1, 1, 1, 1, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
 [0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
 [0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>>
```

次は、ナイトが移動する道筋について考えてみましょう。2 ページで説明した特徴 B により、ナイトは適当な移動回数で盤上のどのマスへも移動できることが分かります。問題は、その移動回数です。これを解析するため、さらに下記の関数 get_mark() を用意することにします。

```
def get_mark(b, x, y):
  o = int((len(b) - 1)/2)          # o,x,y,i,j の対応関係は set_mark() 内のコメントを参照
  j = o + x
  i = o - y
  return(b[i][j])
```

get_mark() は、それぞれ引数として渡した盤配列 b の座標 (x, y) に設定されている数値を返します。動作イメージは下記の通り。

```
>>> b = init_board(1)
>>> set_mark(b, 1, 2, 1)
>>> get_mark(b, 1, 2)        ← 直前で設定した座標 (1,2) の値
1
>>> get_mark(b, 0, 0)        ← 始点 S/座標 (0,0) の値
5
>>> get_mark(b, -2, -3)      ← 座標 (-2,-3) のマスにはまだ何も設定されていないため、初期値 0 が返る。
0
```

課題 1-b: 関数 `init_board()`, `set_mark()`, `get_mark()` を利用し、次の仕様を満たす関数 `count_km()` を作成せよ。

1. 移動回数 n を指定すると、n 回までの移動で到達できるマスを表わした盤配列 `b` を返す。
2. 返値の盤配列 `b` では、i 回 ($i \leq n$) の移動で到達できるマスに値 i が設定されている。
3. また、ナイトの始点 S に該当するマスには数値 0 が設定されている。

`count_km()` の動作イメージは下記の通り。

```
>>> b = count_km(3)          ← ナイトが 3 回以内の移動で到達できるマスと、そこへの移動回数を表示
>>> pprint.pprint(b, width=100)
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 3, 0, 3, 0, 3, 0, 3, 0, 0, 0, 0, 0],
 [0, 0, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 0, 0, 0],
 [0, 0, 3, 0, 3, 2, 3, 2, 3, 2, 3, 0, 3, 0, 0, 0],
 [0, 3, 0, 3, 2, 3, 2, 3, 2, 3, 2, 3, 0, 3, 0, 0],
 [0, 0, 3, 2, 3, 0, 1, 2, 1, 0, 3, 2, 3, 0, 0, 0],
 [0, 3, 0, 3, 2, 1, 2, 3, 2, 1, 2, 3, 0, 3, 0, 0],
 [0, 0, 3, 2, 3, 2, 3, 0, 3, 2, 3, 2, 3, 0, 0, 0],
 [0, 3, 0, 3, 2, 1, 2, 3, 2, 1, 2, 3, 0, 3, 0, 0],
 [0, 0, 3, 2, 3, 0, 1, 2, 1, 0, 3, 2, 3, 0, 0, 0],
 [0, 3, 0, 3, 2, 3, 2, 3, 2, 3, 2, 3, 0, 3, 0, 0],
 [0, 0, 3, 0, 3, 2, 3, 2, 3, 2, 3, 0, 3, 0, 0, 0],
 [0, 0, 0, 3, 0, 3, 0, 3, 0, 3, 0, 3, 0, 0, 0, 0],
 [0, 0, 0, 0, 3, 0, 3, 0, 3, 0, 3, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
>>>
```

ヒント: ナイトは、座標が増える方向にも減る方向にも動けるため、ループを用いて課題 1-b を解くのは難しいです。再帰呼び出しを利用することで、見通しの良いプログラムを作成することができます。

課題 2 — ナイトの周遊路

課題 1 では、ナイトを n 回移動させると ($n \geq 3$)、第 n 八角形内にある始点 S と異なる色のマスに全て到達可能であることが分かりました。この他にも、ナイトの移動に伴う興味深い特徴は数多くあります。その一つに、図 7 の S (四隅の一つ) にあるナイトが、チェス盤上の全マス (チェス盤のサイズは 8 行 × 8 列の計 64 マス) を 1 回ずつ通過する経路は存在するか、という問題があります。もし、このような経路 (これを巡回路と呼ぶことにします) が存在した場合、その解を

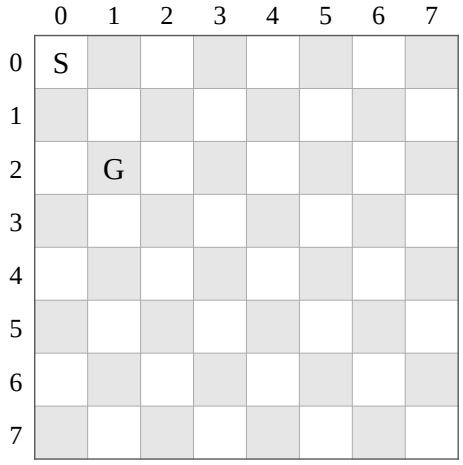

図 7: ナイトの周遊

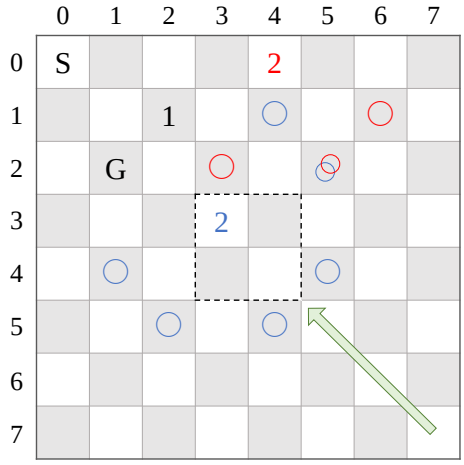

図 8: より最適と想定される次の 1 手

さらに拡張し、最後の 1 手 (63 回目の移動) で G (始点から 1 手先のマス) に到達できる経路を発見できれば、G から 1 手進めると S に至ることから、ナイトはチェス盤上を周遊できることになります (これを周遊路と呼ぶことにします)。周遊なので、S ～ G に至るどのマスを始点にしても、ナイトは盤上の全マスを 1 回ずつ通過して元の位置に戻ることができます。この問題の歴史は大変古く、9 世紀から存在していることが文献より分かっています。また、オイラーやガウスといった数学世界の偉人達も興味を持ちました。

さて、次の課題では、ナイトの周遊路を求めてみることにしましょう。実は、この問題はかなり難しく、計算機を適切に利用しないと周遊路を発見することはほぼできません。とは言え、全く手掛かりがないのか (総当たりで全パターンを調べるしかないのか) と言えば、そうでもありません。図 8 (5 ページ) を例に考えてみます。以下では、i 行 · j 列のマスを座標に似せて < i, j > と表わすことにしましょう (x-y 座標系と配列における i 行 · j 列とを混乱しないように)。

まずは、S から < 1, 2 > へ 1 手動いたとします (マス内には 1 手目であることを示す 1 が表示されています)。次に (2 手目に) 動けるマスは < 0, 4 > および < 3, 3 > ですが、両者を比べてみると、< 0, 4 > から 3 手目に移動できるマス (赤丸) は、< 3, 3 > から 3 手目に移動できるマス (青丸) より少ないことが分かります。ここで、< 0, 4 > と < 3, 3 > のどちらか一方が未到着である将来を考えてみましょう。両者が移動できるマスの数を逆から見ると、< 3, 3 > には将来的に 6 箇所のマスから到達できる一方で、< 0, 4 > には将来的に 3 箇所のマスからしか到達できません。つまり、< 3, 3 > より < 0, 4 > の方が、将来的には到達しづらいことが分かります。将来的な到着が困難であるマスほど、到着できる時に到着しておいた方が手詰まりの発生する可能性は少ないと想定されるので、ここでは < 3, 3 > より < 0, 4 > を選択した方が良さそうだと言えます。

	0	1	2	3	4	5	6	7
0	S	31	2	17	28	33	12	15
1	3	18	29	32	13	16	27	34
2	30	1	42	57	44	53	14	11
3	19	4	45	54	○	26	35	52
4	46	41	58	43	56		10	25
5	5	20	55		59		51	36
6	40	47	22	7	38	49	24	9
7	21	6	39	48	23	8	37	50

図 9: 手詰まりとなる経路

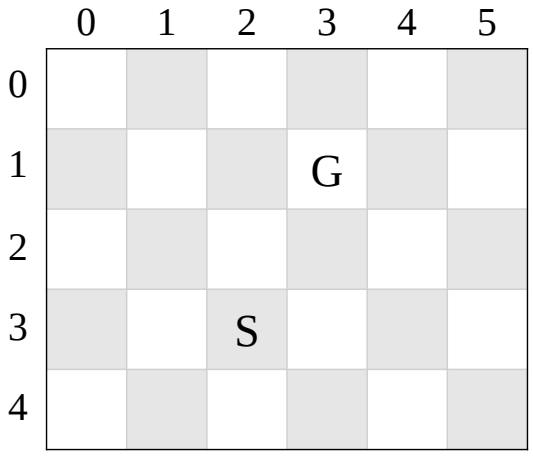

図 10: 5 行 × 6 列のチェス盤

以上より、移動先の候補が複数ある場合は、その優先度は図 8 の緑矢印に沿って低くなる (端のマスが最高で中心に行くにつれ下がる) という選択方針を考えることができます (これを方針 I と呼ぶことにします)。そこで、方針 I に従い、実際にナイトを動かしてみた例を図 9 に示します。各マスの数値はそのマスに至る移動回数です。図 7 のマス G が終点となる経路を最初から発見することは難しいため、ここでは取り敢えずその制約を外した巡回路を求めています。この例では、59 手まで移動できたものの、次の移動先 (赤丸マス) は全て既通過マスであり移動できません。これは、58 手目の < 4, 2 > から 59 手目に < 5, 4 > を選択したことが原因で、59 手目としては < 3, 4 > (青丸マス) を選択すべきでした。とは言え、青丸マスへ移動した後を進めてみるとやはり手詰まりとなるため、さらに戻って 57 手目の < 2, 3 > から 58 手目として < 4, 2 > を選択したことが良くなかったと分かります。もう少し考えてみると、実は 55 手目以降には移動可能なマスがそれぞれ一つしかなく (要は一本道)、複数の移動可能なマスから選択できるようになるには、54 手目の < 3, 3 > まで戻らなければなりません。ここまでの思考 · 実験結果を整理すると、方針 I は合理的のように見えましたが、これだけでは行き詰まる場合があり、その時は別の選択肢が存在するようになるまで (今回の例では別の移動先を選べるようになるまで) 逆戻りする必要があります。この逆戻りを方針 II と呼ぶことにしましょう。

実は、方針 I · 方針 II はそれぞれ、**ヒューリスティク法** (Heuristic Method) および**バックトラック法** (Backtrack Method) と呼ばれています。計算機で扱う問題の一つに、大量の選択肢から条件に合致する解を探し出すという問題があります (文字通り探索問題と呼ばれています)。数値解析のように、効率的に解を求める一般化されたアルゴリズム (例えばオイラー法など) が存在すれば、それを実行すればよいのですが、探索問題の多くにはそのようなアルゴリズムが存在しませ

ん。よって、解の候補を総当たりで調べることになります。しかし、これでは計算量の観点から実用的ではないため、問題の構造や制約の特徴より少しでも効率が良さそうなアルゴリズムを発見して行く (heuristic) わけです。但し、あくまで “良さそうな” というレベルなので、必ず解を探せるわけではありません。そこで、これ以上探索を進めても (アルゴリズムを進めても) 解は存在しないことが判明したら、その探索を終了して別の解候補を選択できるまで戻り (backtrack)、新たな探索を開始する必要があります。課題 1-b のヒントにも書きましたが、戻るという操作が入る場合、一般にループを用いたアルゴリズムは複雑となり、再帰呼び出しを利用した方が見通しは良くなります。

今回は、8 行 × 8 列のチェス盤ではなく、より小型な 5 行 × 6 列の盤 (図 10 · 前ページ) でナイトの周遊路を求めることにします[5]。周遊路は、巡回路のうち始点と終点が図 7 (5 ページ) のような位置関係になっているものです。そこで、まずは巡回路を求めることにしましょう。ナイトの巡回路をヒューリスティク法で探すために、i 行 · j 列のマス < i, j > に対して優先順位を決める下記の関数 `chk_pri()` を用意しました。

```
def chk_pri(i, j):
  priority = 4
  if ((i == 0) or (j == 0) or (i == 4) or (j == 5)):
    priority = 1
  elif ((i == 1) or (j == 1) or (i == 3) or (j == 4)):
    priority = 2
  elif ((i == 2) or (j == 2) or (i == 2) or (j == 3)):
    priority = 3
  return(priority)
```

`chk_pri()` では、盤上に存在するマス ($0 \leq i \leq 4, 0 \leq j \leq 5$ の < i, j >) に対し、$1 \sim 3$ の優先順位 (`priority`) を割り当てます。盤上に存在しないマス ($0 \leq i \leq 4, 0 \leq j \leq 5$ 以外の < i, j >) に対しては、最低の優先順位 4 を割り当てます。`chk_pri()` を利用して、再起呼び出しを用いたヒューリスティク法 + バックトラック法によるナイトの巡回路を探す手順の一例は、以下のようになります。

1: 移動回数が 29 回に達したら**探索成功**を返す。

2: 移動回数が 28 回以下の場合

- 2-A: 移動先の候補とその優先度を決める (ここで `chk_pri()` を利用)。
- 2-B: 移動先の候補が存在する場合 (各候補の優先度が全て 4 の場合は、移動先がないと判断する)
  - 2-B-a: 移動回数`+1` & 最優先の候補 (マス) に移動回数を記録し、移動させる (再起呼び出し)。
  - 2-B-b: 再起呼び出しから**探索成功**を返された場合、選択した候補は巡回路を発見できたので**探索成功**を返す。
  - 2-B-c: 再起呼び出しから**探索失敗**を返された場合、選択した候補では巡回路を発見できないので最低優先度 (優先度 4) を設定し、手順 2-B へ戻る。
- 2-C: 移動先の候補が存在しない場合
  - 2-C-a: 移動回数`-1` & **探索失敗**を返す。

周回路を発見するには、上の手順で発見した巡回路の始点と終点が図 7 (5 ページ) のような位置関係になっているかどうかを調べます。もし、そのような位置関係になっていなかった場合は、巡回路を発見できたとしても探索失敗として扱います。

[5]ナイトの周遊路は、ゲームで利用する通常のチェス盤 (8 行 × 8 列) より小さい盤にも存在することが知られています。8 行 × 8 列では計算量も膨大となり、デバッグも一苦労なので、今回は計算量の少ない小型の盤にしました。本来ならば、5 行 × 6 列といった変則的な盤ではなく、5 行 × 5 列のような規則的な盤を採用したいところですが、この盤にはナイトの周遊路が存在しません。その理由ですが、1 ページで述べたように、ナイトは移動する度に移動先となるマスの色が変わります。5 行 × 5 列の盤はマス数が奇数個であり、巡回路の移動回数は偶数回となるため、例えば白マスを始点とする巡回路の終点は必ず白マスになってしまいます。よって、どのような巡回路であろうと、始点と終点は絶対に図 7 (5 ページ) のような位置関係 (つまり、少なくとも始点と終点の色が異なる) になれないからです。では、4 行 × 4 列の盤ならばどうか、という疑問が湧いて来ますね。実は、4 行 × 4 列の盤には、周遊路どころか巡回路すら存在しないことが分かっています。今回のヒューリスティク法と同じように考えます。図 10 (6 ページ) を参考に、4 行 × 4 列の盤を思い浮かべてみて下さい。このサイズの盤では、隅からは中央 4 マスに含まれる 2 マスのどちらかにしか行き来できません。ここで、左上隅 a と右下隅 b の対 A を考えると、どちらかの隅が始点 or 終点でなければ将来的に隅から移動できないことが分かります (例えば、中央 → a → 中央 → b ⇝ 中央は 2 マスとも既に通過済みなので行先なし)。もう片方の対 B も事情は同じなので、この盤では隅が始点および終点にならざるを得ません。この時、例えば隅 a を始点として対 A を出た後、対 B に入る前に全辺マスの移動を終える必要があります (∴ 対 B には終点があり、“対 B に入る = 終了” となってしまうので)。しかしながら、このような移動はできません (確認してみて下さい)。このように、盤のサイズと周遊路の有無との関係は大変興味深く、マスを頂点に、経路を辺に置き換えたグラフ理論として深く研究されています。

課題 2: 関数 chk_pri() を利用し、次の仕様を満たす関数 knight_tour() を作成せよ。

1. ナイトの初期位置 < i, j > を受け取り、図 10 (6 ページ) に示す 5 行 × 6 列の盤を表わした盤配列 b を返す。
2. 返値の盤配列 b では、< i, j > から t 回の移動で到達できるマスに値 t が設定されている。
3. ナイトの初期位置に該当するマス < i, j > には数値 0 が設定されている。
4. 仕様 2, 3 が示すナイトの経路は巡回路になっている。
5. 仕様 4 が示すナイトの経路は周回路になっている。

knight_tour() の動作イメージは下記の通り。ナイトの始点 (0 のマス) と終点 (29 のマス) が、図 7 (5 ページ) のような位置関係になっていますね。

```
>>> b = knight_tour(2, 2)          ← ナイトの初期位置として<2,2>を指定
>>> pprint.pprint(b, width=100)
[[10, 27, 16, 1, 12, 25],
 [17, 6, 11, 26, 21, 2],
 [28, 9, 0, 15, 24, 13],
 [5, 18, 7, 22, 3, 20],
 [8, 29, 4, 19, 14, 23]]
>>>
```

ヒント: 実際にプログラムを動かしてみると分かりますが、実はナイトの初期位置により巡回路を発見する時間は大きく変わります。これは、ナイトの初期位置に応じてバックトラックの回数が変わるからです。私が試しに作成したプログラムでは、上記の周遊路を発見するまでに要した時間は 3.47 秒でした。また、初期位置を < 2, 3 > に変えてみると、0.81 秒の所要時間で周遊路を発見できました (バックトラックは約 1 万回)。しかし、初期位置 < 0, 1 > では、280 万回以上のバックトラックを行ない、216 秒を要しました。経験的に、初期位置は中央周辺が良さそうに見えます。デバッグの際は参考にしてみて下さい。

課題 1/2 共に、各仕様を完備していない場合でも提出は受理します。

＊ 但し、各関数 init_board(), set_mark(), get_mark(), chk_pri() は、意味のある利用をして下さい (単に呼び出しただけ等、意味のない利用をした場合は、仕様の前提を満たしていないとして扱います)。また、これらの関数を改造してもよいですが、これも意味のある改造/利用をして下さい。