Algorithm Final Project

— *Hidato Puzzle Generator / Verifier / Solver*

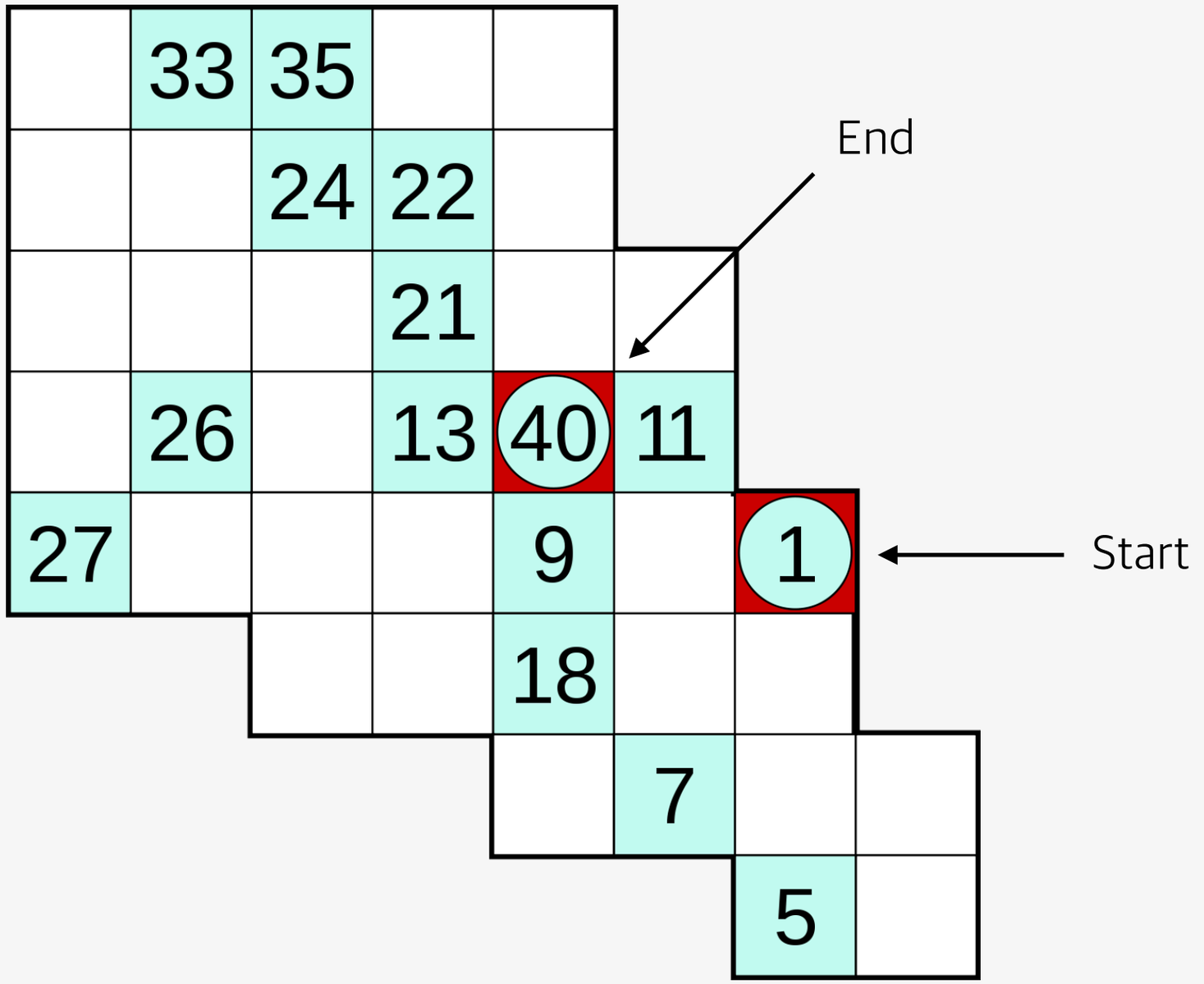
ALGOLINGO & DOPAMINE

국민대학교 소프트웨어융합대학

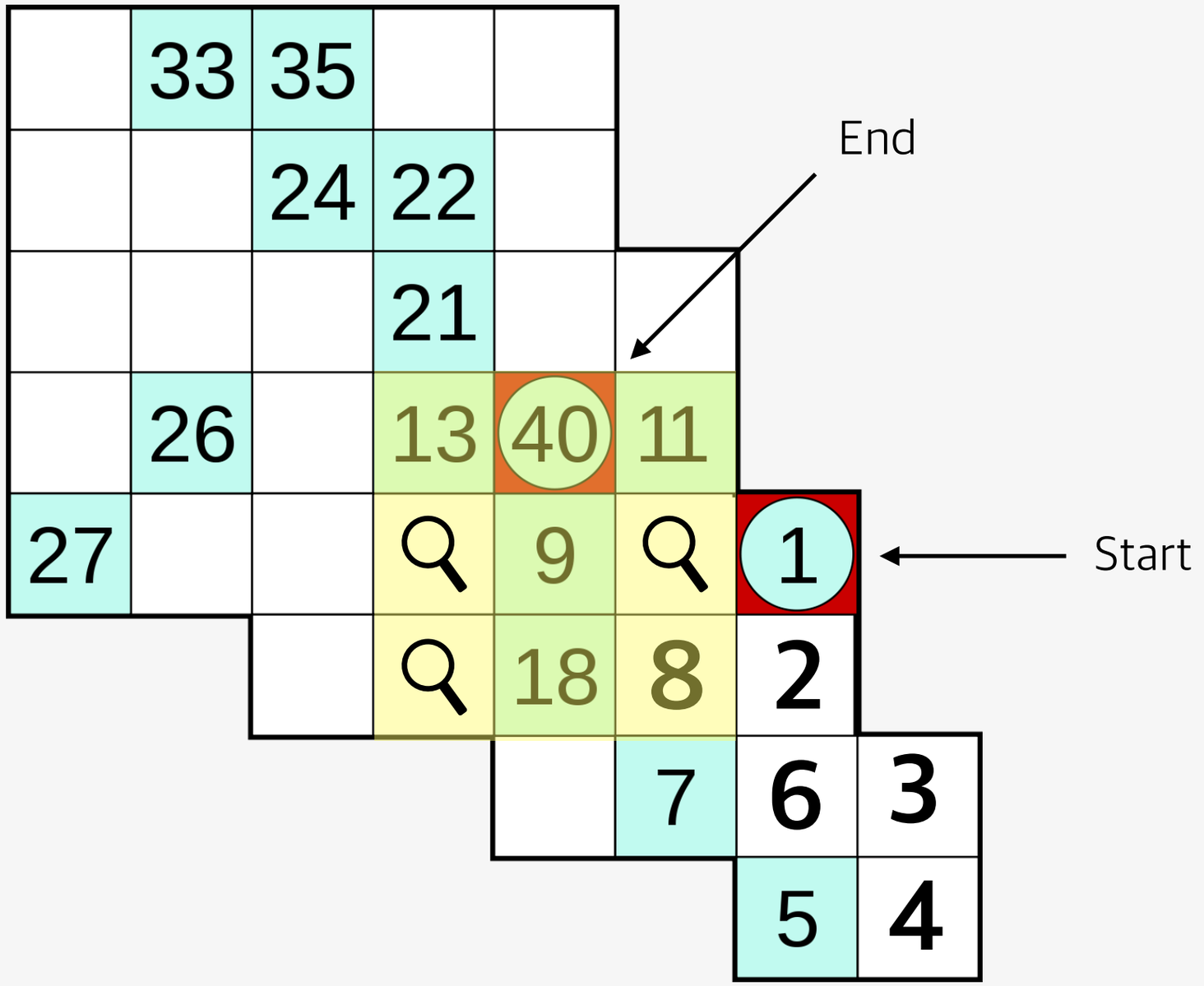
서준교 | 박병훈 | 김상민 | **홍승환** | 이성재 | 최찬경

알고리즘 — 최준수 교수님

Hidato Puzzle — Introduction



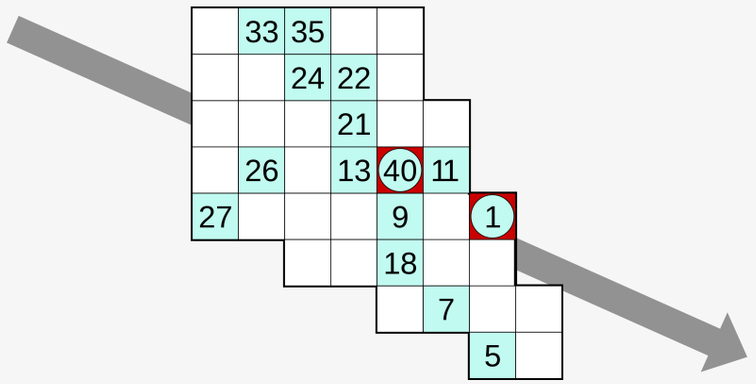
Hidato Puzzle — Introduction



Hidato Puzzle — Project Structure

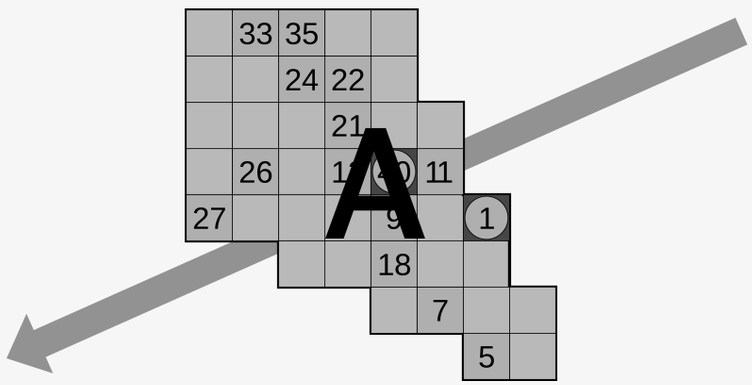
Generator

Input — Given Map
Output — Problem Map



Solver

Input — Problem Map
Output — Answer Map



Verifier

Input — Answer Map
Output — Verification Result

Generator — Input

```
1  ——  입력 데이터의 길이
8  8  ——  한 데이터의 가로 길이 / 세로 길이
0  0  0  0  0  -1  -1  -1
0  0  0  0  0  -1  -1  -1
0  0  0  0  0  0  -1  -1
0  0  0  0  0  0  -1  -1
0  0  0  0  0  0  0  -1
-1  -1  0  0  0  0  0  -1
-1  -1  -1  -1  0  0  0  0
-1  -1  -1  -1  -1  -1  0  0
```

—— 한 데이터의 전체 Map 데이터

Generator — Input

```
1 ── 입력 데이터의 길이
8 8 ── 한 데이터의 가로 길이 / 세로 길이
0 0 0 0 0 -1 -1 -1
0 0 0 0 0 -1 -1 -1
0 0 0 0 0 0 -1 -1
0 0 0 0 0 0 -1 -1
0 0 0 0 0 0 0 -1
-1 -1 0 0 0 0 0 -1
-1 -1 -1 -1 0 0 0 0
-1 -1 -1 -1 -1 -1 0 0
```

── 한 데이터의 전체 Map 데이터

1차원 배열로 저장하여 접근

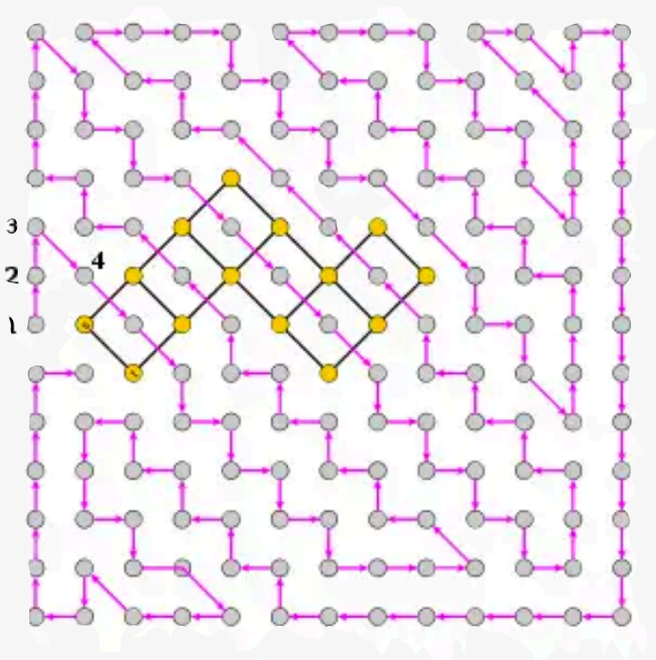
```
int[] map = {0, 0, 0, 0, 0, -1, … };
```

Generator — Algorithm

Hidato is NP-Complete

Hamiltonian Cycle Problem으로 축소하여 증명 가능함
Ex) Knight's Tour Problem

Hidato Game which has a solution (좌측)
if and only if
the starting graph has an Hamiltonian cycle (우측)



33	66	67	68	69	94	95	96	97	107	108	110	111
32	34	65	64	70	71	93	92	98	99	106	109	112
31	35	36	63	62	72	73	91	90	100	101	105	113
30	29	37	38		61	74	75	89	88	102	104	114
3	28	27		39		60		76	87	86	103	115
2	4		26		40		59		77	85	84	116
1	169	5		25		41		58	78	79	83	117
154	155		6	24	23		42	57	56	80	82	118
153	138	137	7	8	22	21	43	44	55	54	81	119
152	139	136	135	9	10	20	19	45	46	53	52	120
151	140	141	134	133	11	12	18	17	47	48	51	121
150	147	142	143	132	131	13	14	15	16	49	50	122
149	148	146	145	144	130	129	128	127	126	125	124	123

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 하나를 무작위로 선택>
  int backupValue = nextData[index]
  nextData[index] = <이웃 칸 중 하나를 무작위로 선택>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int minCount = currentCount
  int diff = currentCount - prevCount

  If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
      If currentCount <= 0 Then Break

nextData = minNextData
```

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 랜덤 선택>
  int backupValue = nextData[index]
  nextData[index] = <이곳에 임시로 숫자 채워넣기>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int minCOunt = currentCount

  int diff = currentCount - prevCount

  If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
      If currentCount <= 0 Then Break

nextData = minNextData
```

무한히 값을 찾는 것을 방지하기 위해서
한계값을 지정함 — 100,000번

일반적으로 10,000번 안에 종료됨

이 과정을 10번 시도 후 발견되지 않았으면
생성에 실패했다는 메시지를 보임
— 지금까지 실패한 사례 없음

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 하나를 무작위로 선택>
  int backupValue = nextData[index]
  nextData[index] = <이웃 칸 중 하나를 무작위로 선택>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int diff = currentCount - prevCount
  If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
      If currentCount <= 0 Then Break

nextData = minNextData
```

nextData를 지속적으로 발전시키면서
빈 칸이 없을 때까지 최대한 반복하는 형태의 알고리즘

NP-Complete 문제이기 때문에
지속적으로 알고리즘에 무작위성이 개입함

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 하나를 무작위로 선택>
  int backupValue = nextData[index]
  nextData[index] = <이웃 칸 중 하나를 무작위로 선택>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int minCount = currentCount
  int diff = currentCount - prevCount
  If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
      If currentCount <= 0 Then Break

nextData = minNextData
```

빈 칸의 개수를 지속적으로 추적하기 위한 변수들

prevCount를 사용하여 이전 빈 칸과의 차잇값을 계산

minCount를 사용하여 최솟값을 지속적으로 갱신함

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 하나를 무작위로 선택>
  int backupValue = nextData[index]
  nextData[index] = <이웃 칸 중 하나를 무작위로 선택>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int minCOunt = currentCount

  int diff = currentCount - prevCount

  If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
    If currentCount <= 0 Then Break

nextData = minNextData
```

지금 값과 이전 값의 차이를 계산함

만약 차이가 양수면 이동 결과가 이전보다 좋지 않다는 의미
backupValue를 활용해서 되돌림

아니라면 지금 결과를 반영함

Generator — Algorithm

```
For i = 0 To 100000
  int index = <빈 공간 중 하나를 무작위로 선택>
  int backupValue = nextData[index]
  nextData[index] = <이웃 칸 중 하나를 무작위로 선택>
  int currentCount = <nextData의 빈 칸에 값을 채워넣고 남은 빈 칸의 개수>
  int prevCount = currentCount
  int minCOunt = currentCount
  If currentCount > prevCount Then
    If diff > 0 Then nextData[index] = backupValue
  Else
    prevCount = currentCount
    If currentCount < minCount Then
      minCount = currentCount
      minNextData = nextData
      If currentCount <= 0 Then Break

nextData = minNextData
```

만약 최솟값보다도 지금 결과가 나은 결과라면
현재의 데이터로 학습 과정을 갱신함

만약 0개로 수렴했다면 최적값을 찾은 것이므로 Break

Generator — Result

```
generate start
            . 32 40  .  .
           30  . 34  .  .
           28  . 35  . 11 10
          .  . 20 13  .  9
          . 24 21  . 15  .  8
                .  .  .  .  6
                      17  5  1  .
                             .  2
generate end
elapsed time: 0.268478 sec
```

Generator — Result

```
generate start
          .   . 136   .   .   . 128 126   .   .   . 109   . 106 105
          . 137   . 134   .   .   . 125   . 114 116 117 107   .   .
          .   .   .   .   .   . 124   .   . 115   .   .   . 100 103
          . 171 172 147 146   . 151   .   . 120   .   .  94   .   .
        170   .   .   .   . 199   . 201   .   .   . 218   .  98  96
        169   . 174 157   .   . 200   . 212   . 221   .  89  90  97
          .   . 158   . 160 197   . 211 203 222 223   .   .  82  81
        177   .   . 164 161   .   . 210 209   .   .  87  86   .   .
        178 166 180   . 163   . 193  40 205   . 225  85   .   .   .
          1 179   . 189   . 192   .   .   .   .   .  73  74  78  76
          2   .   . 187 188   .   .  42  45  46   .   .  72  75   .
          .   5   . 183   .   .  35   .   .  44  53   .  50   .  69
          6   7   .   .  19  21  34  33  31   .   .  54  51  67   .
          .  11  13  14  18   .  23  25   .  56   .   .  62  66   .
          9   .   .  15  16  17   .  26   .  28   .   .   .   .   .
generate end
elapsed time: 0.615928 sec
```

Solver — Algorithm

DFS (Depth First Search)

함수를 재귀적으로 호출하여 길을 탐색함

최종 Leaf에 도달했을 시 정답, 끝이 아닐 시 Backtracking

```
bool search(int x, int y, int nextPoint)
```

Solver — Algorithm

```
If nextPoint == max + 1 Then Return True

If isCheckArr[nextPoint] is True Then
 // (1) 다음 Index 칸에 값이 있을 때
Else
 // (2) 다음 Index 칸에 값이 없을 때
```

Solver — Algorithm (1)

```
// (1) 다음 Index 칸에 값이 있을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue

  If (arr[row_tmp + col_tmp * row] == nextPoint)
  Then If search(row_tmp, col_tmp, nextPoint + 1) is True
    Then Return True

Return False
```

Solver — Algorithm (1)

```
// (1) 다음 Index 칸에 값이 있을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue

  If (arr[row_tmp + col_tmp * row] == nextPoint)
  Then If search(row_tmp, col_tmp, nextPoint + 1) is True
   Then Return True

Return False
```

도달 가능한 8 방향의 칸 각각에 대하여
각 칸의 Row, Column 값을 구하여 가져옴

dx, dy에는 8 방향의 칸에 대한 좌표가 저장됨

Solver — Algorithm (1)

```
// (1) 다음 Index 칸에 값이 있을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue

  If (arr[row_tmp][col_tmp] == nextPoint)
  Then If (DFS(row_tmp, col_tmp, nextPoint + 1) is True
     Then Return True

Return False
```

만약 Row Size나 Column Size가 초과되면
이 칸에 대한 검사를 수행하지 않음

Solver — Algorithm (1)

```
// (1) 다음 Index 칸에 값이 있을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue
```

```
If (arr[row_tmp + col_tmp * row] == nextPoint)
Then If search(row_tmp, col_tmp, nextPoint + 1) is True
  Then Return True
```

Return False

만약 다음 칸에 씌어있는 숫자가 nextPoint와 같을 경우
— 즉 옳은 길로 온 경우

여기에서부터 다시 재귀하여 지속적으로 탐색을 수행함

이 결과가 True라는 것은 최종 결과를 찾았다는 의미이므로
다시 스택 위로 True를 Return하여 올림

Solver — Algorithm (1)

```
// (1) 다음 Index 칸에 값이 있을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue

  If (arr[row_tmp + col_tmp * row] == nextPoint)
  Then If search(row_tmp, col_tmp, nextPoint + 1) is True
   Then Return True

 Return False
```

이 부분에 도달했다는 것은 잘못된 길에 들어섰음을 의미하므로
False를 Return하여 이 사실을 호출 측에 알림

Solver — Algorithm (2)

```
// (2) 다음 Index 칸에 값이 없을 때
For i = 0 To 8
  int row_tmp = x + dx[i]
  int col_tmp = y + dy[i]

  If row_tmp < 0 || row_tmp >= row
     || col_tmp < 0 || col_tmp >= col
  Then Continue

  If (arr[row_tmp + col_tmp * row] == 0)
  Then
    arr[row_tmp + col_tmp * row] = nextPoint;

    If search(row_tmp, col_tmp, nextPoint + 1) is True
    Then Return True

    arr[tow_tmp + col_tmp * row] = 0;

Return False
```

Solver — Algorithm (2)

```
// (2) 다음 Index 칸에 값이 없을 때
만약 지정된 위치가 0을 가졌다면
— 탐색한 칸이 빈 칸이라면
다음에 나올 숫자 값을 써넣고 이 값을 기준으로 재귀함
재귀 결과 옳은 길이 아닐 경우 다시 빈 칸으로 만들고 다음 칸을 탐색함
— Return True에 닿지 못했다는 것은 옳지 않은 길이라는 뜻
```

```
If (arr[row_tmp + col_tmp * row] == 0)
Then
  arr[row_tmp + col_tmp * row] = nextPoint;

  If search(row_tmp, col_tmp, nextPoint + 1) is True
  Then Return True

  arr[tow_tmp + col_tmp * row] = 0;
```

```
Return False
```

Solver — Result

```
10 10
-1 -1 -1 -1 0 53 -1 -1 -1 -1
-1 -1 -1 -1 0 0 -1 -1 -1 -1
-1 -1 56 0 0 0 30 0 -1 -1
-1 -1 0 0 0 0 0 0 -1 -1
-1 0 0 20 22 0 0 0 0 -1
-1 13 0 23 47 0 41 0 34 -1
0 0 11 18 0 0 0 42 35 37
0 0 0 0 5 3 1 0 0 0
-1 -1 -1 -1 0 0 -1 -1 -1 -1
-1 -1 -1 -1 7 0 -1 -1 -1 -1
```

Solver — Result

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

Verifier — Algorithm

두 가지 요건의 만족 여부를 검사

(1) 문제 데이터와 답안을 비교하여 다른 부분이 없어야 함

(2) 시작 위치에서 끝 위치까지 정상적으로 길이 만들어져야 함

```
int verify(int[] solve, int width, int height)
```

Verifier — Algorithm (1)

```
// (1) 문제 데이터와 답안을 비교하여 다른 부분이 없어야 함
For i = 0 To width * height
  If problem[i] != 0 And problem[i] != solve[i]
  Then Return -1
```

Verifier — Algorithm (1)

전체 데이터에 대하여
문제에서 제시한 빈 칸이 아닌 칸 전부를 검사해서
Solver가 이 부분에 수정을 가하지 않았는지를 확인함
수정을 가했다면 -1을 반환해 검증 실패를 알림

```
// (1) 문제 데이터와 답안을 비교하여 다른 부분이 없어야 함
For i = 0 To width * height
  If problem[i] != 0 And problem[i] != solve[i]
  Then Return -1
```

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x

            If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
               solve[di] != checkNumber Then
                count = count + 1
                Continue

            checkNumber = checkNumber + 1
            index = di
            Escape from the nested for loop;

If count >= 8 Then
  return -2
```

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]
```

시작 위치과 끝 위치를 찾음

시작 위치는 index 변수에 넣어서 탐색 시작에 사용

끝 위치는 lastNumber에 넣어서 탐색을 마치는 조건으로 사용

```
While checkNumber <= lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x

            If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
               solve[di] != checkNumber Then
                count = count + 1
                Continue

            checkNumber = checkNumber + 1
            index = di
            Escape from the nested for loop;

If count >= 8 Then
  return -2
```

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x
            If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
            ss[di] == checkNumber Then
                count = count + 1
                Continue

            checkNumber = checkNumber + 1
            index = di
            Escape from the nested for loop;

 If count >= 8 Then
   return -2
```

checkNumber가 lastNumber와 같지 않은 동안 반복함
— 현재 탐색하는 숫자가 끝 숫자랑 같아지면 탐색을 종료함

-1부터 1까지 반복하는 2중첩 For문을 만듦
— 이 두 수를 조합해 8방향 칸의 Index에 대한 접근을 만듦

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x
            If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
               solve[di] != checkNumber Then
                count = count + 1
                Continue

            checkNumber = checkNumber + 1
            index = di
            Escape from the nested for loop;

If count >= 8 Then
  return -2
```

현재 Index와 i, j를 사용해서 X, Y 좌표를 계산함
X, Y는 2차원 좌표이므로 배열 정보 추출을 위해 1차원으로 바꿈

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
     int count = 0
     For i = -1 To 1
          For j = -1 To 1
               If i == 0 And j == 0 Then Continue
               int x = index % width + j
               int y = index / width + i
               int di = y * width + x
               If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
                  solve[di] != checkNumber
               Then
                   count = count + 1
                   Continue
               checkNumber = checkNumber + 1
               index = di
               Escape from the nested for loop;

If count >= 8 Then
  return -2
```

만약 검사하려는 이웃 칸이 주어진 공간을 벗어났거나
다음 숫자와 일치하지 않을 경우 count 값을 1 증가시킴
— count 값이 8을 넘을 경우 모든 칸을 검사한 것으로 확인하려는 목적

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x
            If x < 0 Or y < 0 Or x >= width Or y >= height Or
               solve[di] != checkNumber Then
                count = count + 1
            Else
                checkNumber = checkNumber + 1
                index = di
                Escape from the nested for loop;

If count >= 8 Then
  return -2
```

여기에 도달했다는 것은 다음에 올 번호를 찾았다는 것을 의미함

checkNumber를 1 올려주고 현재 Index를 갱신함

2중 For문을 탈출하여 다음 번호로 이동하게 함

Verifier — Algorithm (2)

```
For i = 0 To width * height
  If solve[i] == 1 Then index = i
  If solve[i] > lastNumber Then lastNumber = solve[i]

While checkNumber != lastNumber
    int count = 0
    For i = -1 To 1
        For j = -1 To 1
            If i == 0 And j == 0 Then Continue
            int x = index % width + j
            int y = index / width + i
            int di = y * width + x

            If x > width - 1 Or x < 0 Or y > height - 1 Or y < 0 Or
               solve[di] != checkNumber Then
                count = count + 1
                Continue
            checkNumber = checkNumber + 1
            index = di
            Break  // Exit the nested for loop;
```

만약 count 값이 8 이상이 되었을 경우
— 8 방향 칸을 모두 검사했으나 찾지 못한 경우
-2를 반환하여 검증 실패를 알림

```
If count >= 8 Then
  Return -2
```

Conclusion

성공적으로 동작하는 Hidato Puzzle Generator / Solver / Verifier를 만듬

> 선형 시간에 해결 불가능하였으나 빠른 시간 안에 해답을 도출함

전체 구성을 모듈화하여 일반화된 형태로 개발함

> 다른 곳에서 필요한 경우 쉽게 사용할 수 있음

Github에 모든 소스와 작업 과정을 업로드하고 공유함

> Google에 “히다토 퍼즐 알고리즘”으로 검색하면 첫 번째 검색 결과

참고문헌

Hidato — 위키백과

> https://en.wikipedia.org/wiki/Hidato

Hidato is NP-Complete — 블로그 글

> http://www.nearly42.org/cstheory/hidato-is-np-complete/

Hamiltonian Path — 위키백과

> https://en.wikipedia.org/wiki/Hamiltonian_path

Thank You

Algorithm Final Project

— Hidato Puzzle Generator / Verifier / Solver

ALGOLINGO & DOPAMINE

국민대학교 소프트웨어융합대학

서준교 | 박병훈 | 김상민 | 홍승환 | 이성재 | 최찬경