미로찾기

2013002545 정광은

코드설명

<전층의 실행방법>

- 바로 실행시키면 output이 나오도록 되어있음
- 전층을 동시에 돌릴 수 있음

```
#first_floor()
#second_floor()
#third_floor()
#fourth_floor()
#fifth_floor()
```

<코드 동작설명>

1. block class 설명

```
class block():

    waiting = []
    expanded = []
    idxInExpanded = -1

    def __init__(self, location):
        self.location = location
        self.row = location[0]
        self.col = location[1]
        self.heuristic = 10000000000000000
        # 확장하면서 heuristic이 계산되기 때문에
        # 확장이 되지 않고 만들어지는 최초의 block의 경우
        # min heuristic을 계산하는데 방해되지 않을정도로 큰 휴리스틱을 부여함

        block.waiting.append(self)
        self.myExpandedIdx = None
        self.prevIdx = 1000
        self.numChild = 0
```

```
    # 휴리스틱은 goal까지의 직선거리로 계산
    def calHeuristic(self, goal):
        return ((goal[0] - self.row)**2 + (goal[1] -
self.col)**2 )**0.5

    # 뒤에 설명
    def dfsExpand(self):
        # ……
    def greedyExpand(self, targetIdx, goal):
        # ……
```

[block의 생성]

- start의 block을 제외하고 모든 block은 부모 block이 expand를 해줄 때 생성됨
- 모든 block은 생성되자마자, expand의 후보가 되는 block의 목록인 block.waiting[]에 들어감
- 실제로 expand를 시도한 이후에는 block.expanded[] 에 들어가며, block.waiting[] 에서는 제거됨.
- 이때 내가 block.expanded[]의 어떤 index로 들어갔는지를 나의 자식 block들의 .prevIdx에 기록해둠

- block 은 class 변수로 다음과 같은 정보를 가짐

.waiting = []
→ 확장되길 기다리고 있는 block들을 저장한다.
부모가 자식 block을 생성할 때 자식 block은 자동으로 들어감

.expanded = []
→ 확장이 된 block들을 저장한다

.idxInExpanded = -1
→ 확장을 위한 .dfsExpand(), .greedyExpand()같은 함수가 실행될 때, 확장을 끝내고 나서 방금 노드확장을 한 parent block이 expanded[] 의 몇번째 index로 들어가게 될 것인가를 표시

• block object는 멤버변수로 다음과 같은 정보를 가짐

(1) .row / .col
block의 maze상의 위치정보

(2) .heuristic / .numChild
block의 expand에 필요한정보

.heuristic (Greedy)
해당 block의 heuristic 은 expand 되는순간
calHeuristic(self, goal) 통해 계산됨

.numChild (Dfs)
expand를 막 마친 block은 waiting[]에서 제외되는데,
방금 expand된 child의 개수만큼 뒤에서부터 세서 제거하기 때문에 필요함

(3) .prevIdx
block이 goal일시 optimal path를 찾기 위해,
나를 확장한 부모 block의 expanded[] 에서의 index인 prevIdx 를 갖는다

[block의 expand]

• block class는 각 알고리즘의 expand를 처리하기 위한 각자의 멤버함수를 가지고 있음
(blockObject).dfsExpand()
(blockObject).greedyExpand() 뒤에 설명

2. 함수 설명

[1] .***ExPand() 함수

1) 표시원칙
- 확장하는 길은 '7' 로 표시한다
- '1'도 아니고 '7'도 아닌 경우만 (in ['2','4','6','5'])
expand할 수 있다
- key search와 goal search의 중간에 '7'로 표시된 확장된 block들에 한해 ('5'는 그대로 둠)convert7to2()함수를 이용해 원상태였던 2로 바꿔둔다.
(goal search 하며 key search 에서 확장한 길을 다시 쓸 수 있도록 하기위함)

2) 작동법

① 현재 확장중인 block이 확장 후 .expanded[]의 어떤 index로 들어갈지를 업데이트

② 현재 block 에서 상하좌우를 살펴 확장될 수 있는 노드를 생성한다

③ 내가 expanded[] 의 어디로 들어갈지에 대한 정보를,
방금 내가 확장한 child block의 prev.Idx에 저장

④ 확장했으므로 time += 1

⑤ 확장했으므로 .expanded[]에 넣고, .waiting[] 에서는 뺀다

1) dfsExpand

```
def dfsExpand(self):
    global maze
    global mazeSize
    global time

    block.idxInExpanded += 1
    self.myExpandedIdx = block.idxInExpanded ①

    if(self.row-1 >= 1): ②
        if maze[self.row-1][self.col] in ['2','4','6','5']:
            block([self.row-1,self.col])
            self.numChild += 1
            block.waiting[-1].prevIdx = self.myExpandedIdx ③
            if maze[self.row-1][self.col] != '4':
                maze[self.row-1][self.col] = '7'
            time += 1 ④
    # 나머지 방향 생략

    # 방금 확장한 parent block 의 위치는 dfs 의 경우
    block.waiting[-self.numChild-1] 에 있으므로
    # 이를 block.expanded[] 에 넣고, block.waiting[] 에서는 지운다
    block.expanded.append(block.waiting[-self.numChild-1]) ⑤
    del block.waiting[-self.numChild-1]
```

2) greedyExpand

```
def greedyExpand(self, targetIdx, goal):
    global maze
    global mazeSize
    global time
```

```
# targetIdx 는 block.waiting[] 에서 가장 heuristic이 작은
   block의 block.waiting[]상의 index 임
block.idxInExpanded += 1
self.myExpandedIdx = block.idxInExpanded ①

if(self.row-1 >= 1): ②
    if maze[self.row-1][self.col] in ['2','4','6','5']:
        block([self.row-1,self.col])
        block.waiting[-1].prevIdx = self.myExpandedIdx ③
        block.waiting[-1].heuristic
          = block.waiting[-1].calHeuristic(goal)
        if maze[self.row-1][self.col] != '4':
            maze[self.row-1][self.col] = '7'
        time += 1 ④
# 나머지 방향 생략

block.expanded.append(block.waiting[targetIdx]) ⑤
del block.waiting[targetIdx]
```

[2] (*blockobject).calHeuristic(self, goal):
현재 위치에서 goal과의 직선거리를 계산해 return 함

[3] findOptimalPath(start, targetIdx):

- start : 시작지점의 위치 [row, col]
- targetIdx : expand를 하다가 발견된 goal의 waiting[] 상의 위치
- targetIdx를 통해 block object 자체에 접근할 수 있음
- block object는 부모 block의 expanded[]상의 index인 prevIdx를 가지고 있음
- 이 prevIdx를 통해 부모 block으로 거슬러가며, 이때 찾은 optimal path는 모두 '5' 표시를 해둠
- 거슬러 가다가 start를 만나면 종료
- '5'가 표시될 때마다 length += 1 로 업데이트

```
global maze
global length
tempBlock = block.waiting[targetIdx]
while(tempBlock.location != start):
    tempBlock = block.expanded[tempBlock.prevIdx]
    maze[tempBlock.row][tempBlock.col]='5'
    length += 1
```

[4] idxOfMinHeuristic()

block.waiting[] 안의 block들중 heuristic이 가장 작은 block의 index를 반납함

[5] convert7to2():

- key 를 search 하다보면 확장된 노드들이 '7'로 표시됨
- 이를 findOptimalPath(start, targetIdx) 를 거치고 나면 optimal path들은 '5'로 표시됨.
- convert7to2() 는 7과 5가 뒤섞여 있는 상태의 maze중 7만 2로 바꿈으로써 goal 을 search 할 때 key search 로 확장된 block들 역시 확장가능하도록 해줌

[6] greedy(start, goal):

- start, goal : 각각 시작/목적지의 위치 [row, col]
- 동작

  goal 을 발견할 때까지 .greedyExpand() 함수를 부른다.

  이때 priority는 waiting[] 에 있는 block 들중 가장 heuristic이 작은 block이다.

  (그 블럭의 위치는 idxOfMinHeuristic()를 불러 찾는다.)

- return

  expand를 하다가 goal 을 발견하면, 해당 block 의 waiting 상의 위치인 targetIdx 를 return 한다.

```
def greedy(start, goal):
    block(start)
    targetIdx = 0
    while(len(block.waiting) != 0 and block.waiting[targetIdx].location != goal):
        block.waiting[targetIdx].greedyExpand(targetIdx, goal)
        targetIdx = idxOfMinHeuristic()
    return targetIdx
```

[7] dfs(start, goal):

- 동작

  goal 을 발견할 때까지 .dfsExpand() 함수를 부른다.

  이때 priority는 waiting[] 에 있는 block 들중 가장 마지막에 들어온 block 이다. (block.waiting[-1])

- return

  expand를 하다가 goal 을 발견하면, 해당 block 의 waiting 상의 위치인 -1를 return 한다.

```
def dfs(start, goal): # start : 시작지점위치, goal : 목표지점위치
    block(start)
    while(len(block.waiting) != 0 and block.waiting[-1].location != goal):
        block.waiting[-1].dfsExpand()
    return -1
```

[6] *_floor():

① key를 찾기 위해.dfs 나 .greedy 를 불러서
goal block의 .waiting[] 에서의 index를 targetIdx에 저장한다

② findOptimalPath 함수로 maze에서 optimal path만 '5'로 표시한다

③ convert7to2 함수로 '7'과 '5'중 '7'만 '2'로 바꿔준다

④ '2'와 '5'가 뒤섞인 상태의 maze를 finalMaze에 deepcopy 해준다

⑤ goal 을 찾기 위해 다시한번 1)2)를 반복한다.

⑥ finalMaze에는 key로 가는 과정에서 발견한 '5'가 들어있고
maze에는 goal로 가는 과정에서 발견한 '5'가 '7'이 뒤섞여 있으므로
둘중 하나에서 '5'였던 block을 finalMaze에서 '5'로 바꿔주면
전체 search에 대한 optimal path가 나옴

```
def *_floor():
    global start, key, goal
    global maze, finalMaze
    global mazeSize, length, time

    # 이전 *_floor() 함수를 실행하며 남은 data를 제거
    maze.clear()
    readMaze(maze, "first_floor.txt")
    setKeyElement()
    block.waiting.clear()
    block.expanded.clear()
    block.idxInExpanded = -1

    # *** 은 greedy 또는 dfs 이며,
    targetIdx = ***(start, key) ①

    # targetIdx 에 들어가는 값은 dfs 면 -1,
      greedy면 최소heuristic을 갖는 block의 waiting[] 에서의 index이다
    findOptimalPath(start, targetIdx) ②
    convert7to2() ③
    finalMaze = copy.deepcopy(maze)④
```

```
# goal을 찾기 위해 다시 expand를 해야하기 때문에,
expand와 관련된 모든 data를 초기화 한다
block.waiting.clear()
block.expanded.clear()
block.idxInExpanded = -1

⑤
targetIdx = ***(key, goal)
findOptimalPath(key, targetIdx)

⑥
for i in range(1,mazeSize):
    for j in range(0,mazeSize):
        if maze[i][j]=='5' or finalMaze[i][j]=='5':
            finalMaze[i][j]='5'

# 출발지점이 5로 바뀌게 되어서 3으로 바꿔주고,
  프린트시 위의 maze size와 관련한 정보는 지워주기 위해 0번째 row는 지워준다
finalMaze[start[0]][start[1]] = '3'
del finalMaze[0]
writeMaze(finalMaze, "*_floor_output.txt")
```

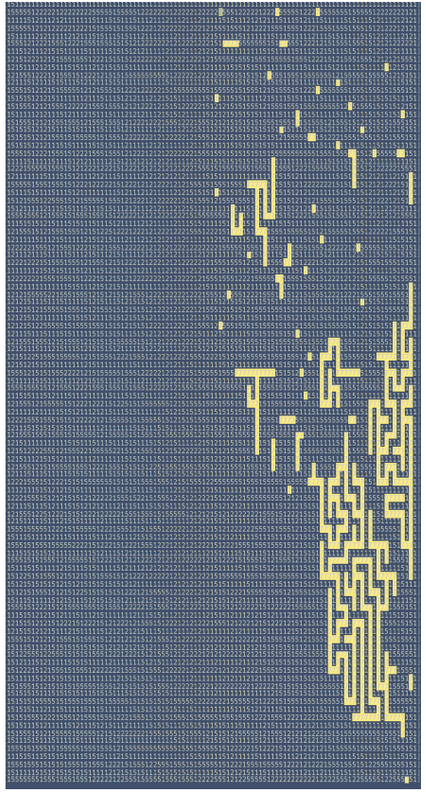
사용 알고리즘

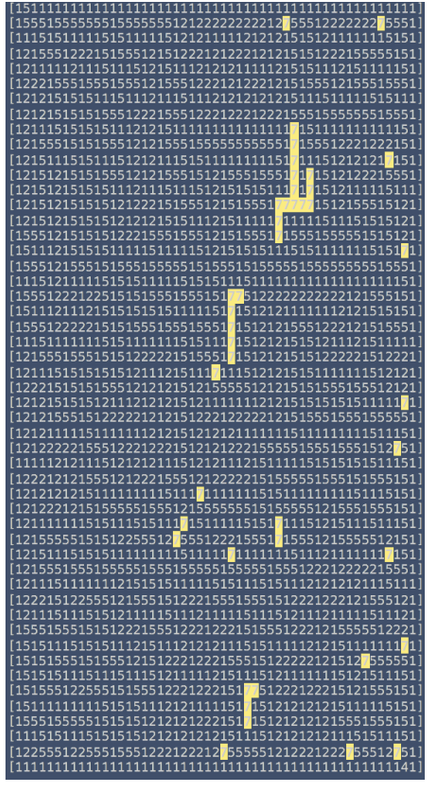
1층, 2층 : DFS

일단 둘다 bfs 나 ucs를 쓰며 느긋하게 확장해서 골을 찾기엔 maze size가 너무 커보였다. 그렇다고 ids 를 쓸 정도로 maze size가 크진 않아서, ids의 반복하게 되는 단점이 두드러질 것 같았다.

이때 maze를 볼 때 dfs 를 쓰기 적절하다고 판단했던 부분은
어떤 block 이 두개의 block을 expand했다 할 때, 바른길과 틀린길중 틀린길이 그다지 깊지 않고 금방 막힌다는 점이였다.즉 dfs 로 잘못된 길을 집입하게되어도 time을 별로 소모하지 않고 다시 바른길로 돌아올 수 있어보였다. 1층은 솔직히 이 경향이 심하진 않았지만 2층에서는 효과를 잘 발휘 했던 것 같다.

(예 : 1,2층에서 key~goal를 찾을 때까지 optimal path보다 더 간 부분을 표시)





3층, 4층, 5층 : Greedy Best Search

사이즈가 작은 만큼 BFS 나 USC도 문제는 없을 것 같았지만
BFS 나 USC는 time을 가장 많이 잡아먹을 걸 거의 보장하는 알고리즘이라 피했다.

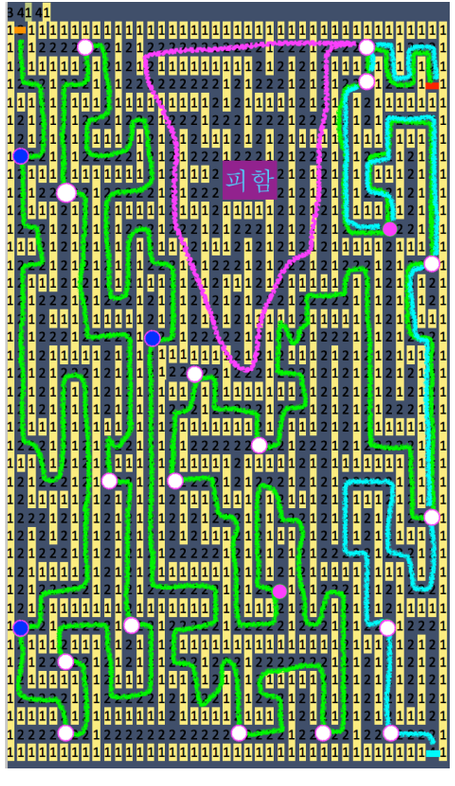
3,4,5층은 1,2층에 비해 DFS로 잘못된 길을 선택할 경우 많이 가야되는 지점들이 보였다.
그리고 이런 지점들은 heuristic을 썼을때 포함한 함수를 쓰면 피할 수 있는 경우가 많았다.(흰색원으로 표시한 부분)

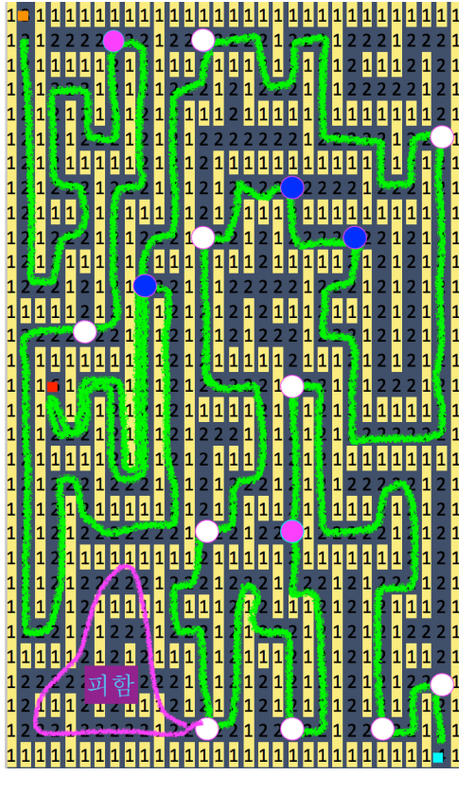
heuristic에 의해 잘못된 길을 들어설 수도 있었지만 (파란색원 부분)그 정도를 비교했을 때 했을때 heuristic을 쓴 경우가 더 이익일 것 같아 heuristic을 포함한 알고리즘인 greedy와 a*을 써보았는데 실제 결과 모두 greedy가 최소로 나와서 이를 적용했다

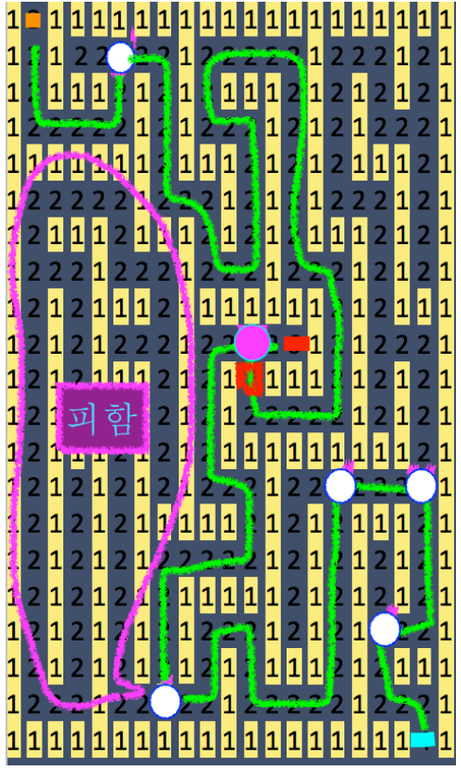
각각 3,4,5 층

white : 휴리스틱사용시 유리, blue : 휴리스틱사용시 불리

magenta : 유불리가 key를 찾느냐 goal을 찾느냐에 따라 다르거나, neutral 한 경우







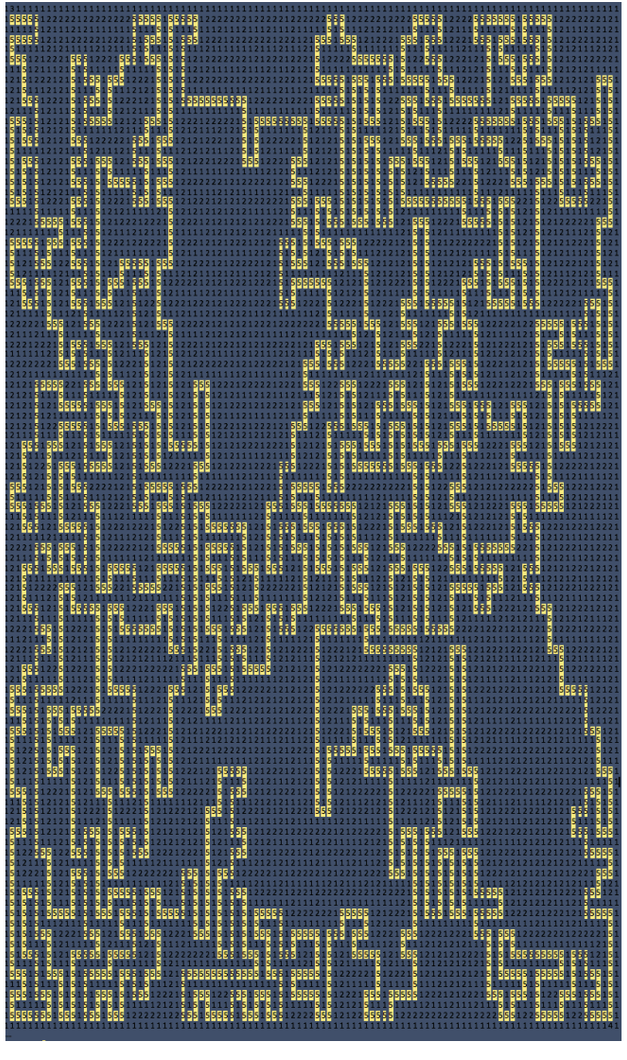
최단경로

탐색한노드의 개수

1층

length = 3850

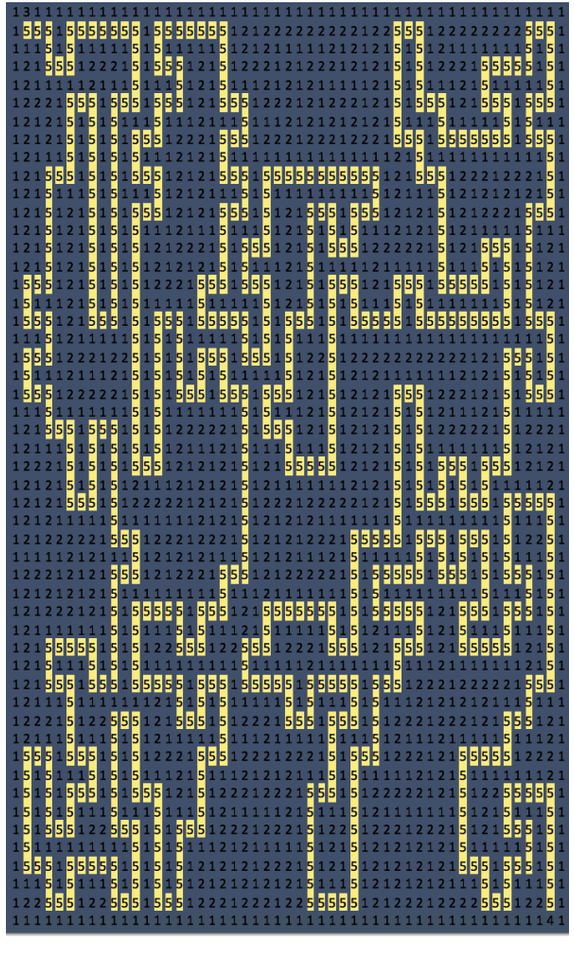
time = 5183



2층

length = 758

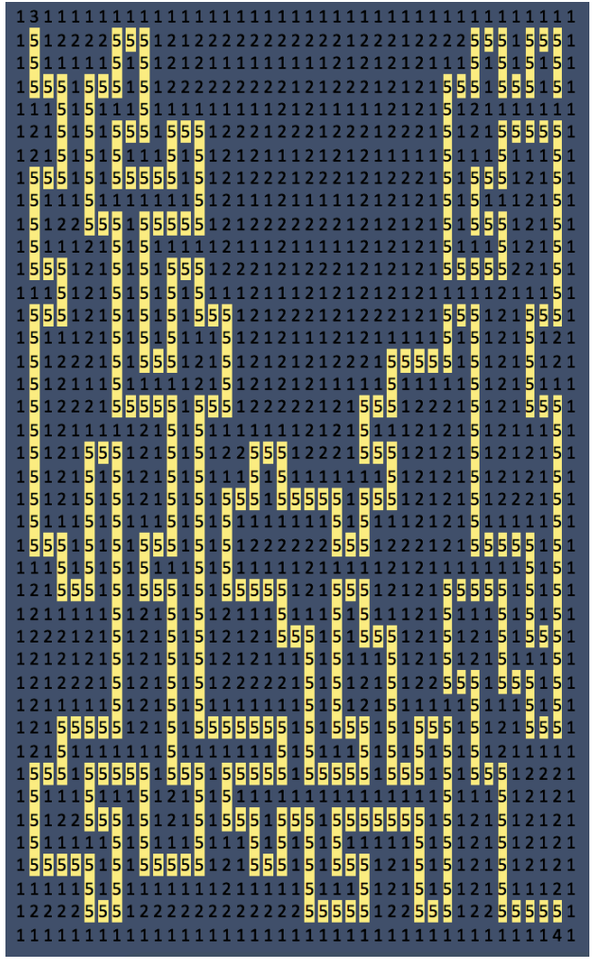
time = 877



3층

length = 554

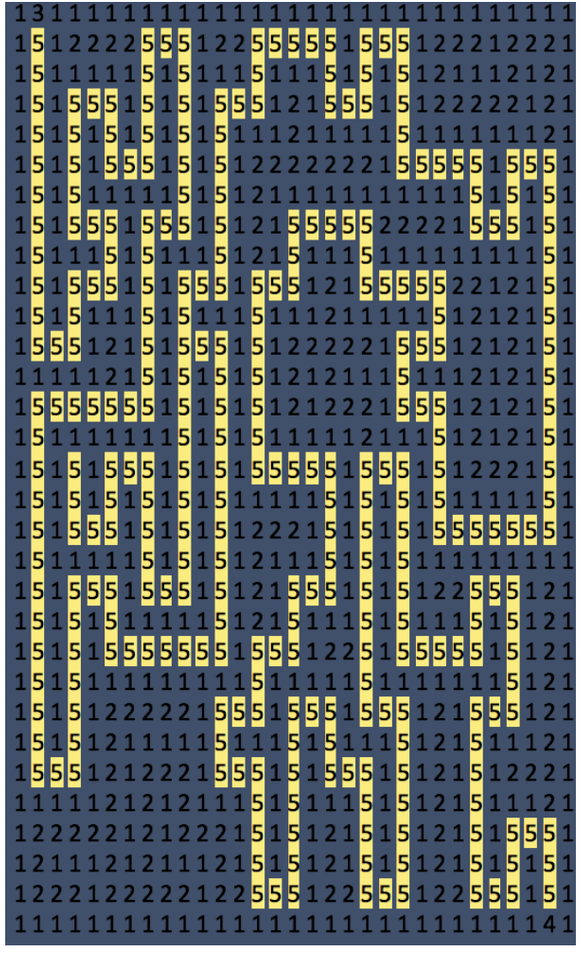
time = 668



4층

length = 334

time = 442



5번 미로

length = 106
time = 126