

20200183 박정원 AI02

1. 사용한 라이브러리와 각 알고리즘마다 구현한 방법에 대한 간단한 설명

(1) 사용한 라이브러리

```
##### Write Your Library Here #####  
from collections import deque, defaultdict  
import heapq  
import math  
import itertools  
#####
```

BFS 구현을 하는 데에 있어 덱 자료구조를 사용하기 위한 collections 의 deque 와
딕셔너리 반환 값을 조정하기 위한 defaultdict

우선순위 큐를 사용하기 위한 heapq 라이브러리

초기화할 때 inf 값을 사용하기 위한 Math 라이브러리

Permutations 를 사용하기 위한 itertools 라이브러리 (astar_four_circles 에서)

(2) 각 알고리즘마다 구현한 방법에 대한 간단한 설명

Stage 1.

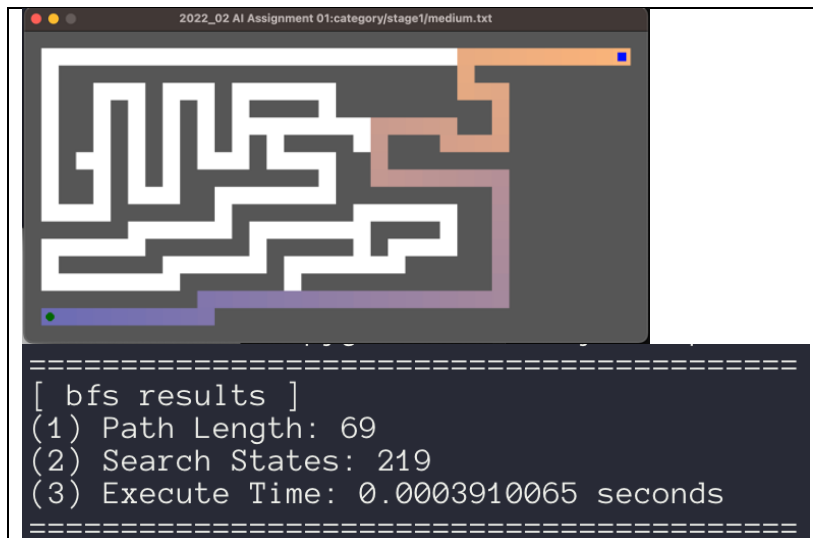
BFS

BFS / Small



```
=====
[ bfs results ]
(1) Path Length: 9
(2) Search States: 15
(3) Execute Time: 0.0000739098 seconds
=====
```

BFS / Medium

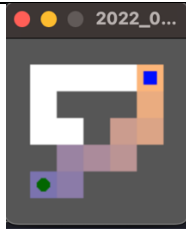


BFS 는 큐 자료구조를 사용해서 구현하였다. 큐에는 좌표가 튜플 형태로 들어가고, 큐가 빌 때까지 while 이 돌도록 했다. 큐에서 꺼낸 현재 위치의 좌표를 (x,y) 라고 할 때, (x,y) 가 목표지점일 경우에는 큐 안에 있는 남은 노드를 탐색할 필요가 없기에 break 를 해주었다. 현재 위치에서 갈 수 있는 동, 서, 남, 북의 위치는 neighborPoints 메소드를 통해서 nxt 라는 리스트로 받았다. 이 nxt 리스트의 좌표들을 보는데, 방문 처리 용도로 선언한 2 차원 visited 리스트(visited[nx][ny]) 를 통해서 방문이 된 노드라면 continue 로 넘어간다. 방문되지 않은 노드라면, 큐에 해당 위치를 넣어주고, 경로 저장을 위해서 선언한 prev 2 차원 리스트에 튜플

형태로 이전 위치인 (x,y)를 저장해준다. (prev[nx][ny] = (x,y)) 또한, visited[nx][ny] = 1 을 통해서 방문 처리를 해준다. BFS 가 종료되고 난 다음에는 경로를 path 리스트에 추가하기 위해서 목표 지점에 도착했을 때 해당 좌표를 gx, gy 에 저장해두었는데, 이를 사용하여 출발 지점까지 반대로 좌표를 추가해준다. 마지막에는 출발 지점을 추가해준 다음에 path.reverse() 를 통해서 역순으로 저장된 경로를 뒤집어주어 최단 경로를 만들어준다.

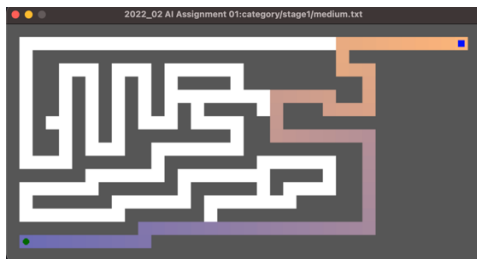
ids

IDS / Small



```
[ ids results ]
(1) Path Length: 9
(2) Search States: 57
(3) Execute Time: 0.0001788139 seconds
```

IDS / Medium



```
[ ids results ]
(1) Path Length: 69
(2) Search States: 8609
(3) Execute Time: 0.0203490257 seconds
```

IDS / Big



ids 는 목표 지점을 찾을 때까지, 최대 탐색 깊이를 0 부터 1 씩 증가시키면서, DFS 를 호출하는 방식으로 구현하였다. 이는 dls, 즉 depth limited search 라고 할 수 있다. Ids 함수 내부에 dls 함수를 선언해서 사용할 수 있도록 하였다. Dls 함수는 start, limit, k 를 인자로 받는데, start 는 시작 위치를 갖는 튜플이고, limit 은 최대 탐색 깊이, k 는 현재 깊이를 의미한다. 그 이후의 구현 방식은 DFS 의 구조와 크게 다르지 않다. 현재 위치가 (x,y)에 저장되어 있고, 목표지점을 찾았는지 확인하기 위해 found Boolean 변수를 선언해준다. Dls 는 현재 깊이를 나타내는 K 가 limit 보다 작을 때, neighborPoints 를 통해서 다음 위치의 후보들을 nxt 리스트에 받아주고, 이 리스트를 통해 후보 위치를 nx, ny 를 통해 받는다. 이 때, 해당 위치까지의 이동 거리를 저장한 visited 2 차원 리스트를 통해서 다음 노드의 visited[nx][ny] 가 visited[x][y] + 1 보다 작거나 같다면 continue 를 통해서 다음 후보 위치로 넘어가도록 해준다. 이를 해주는 이유는, DFS 가 더 먼 경로를 통해서 특정 지점을 지나갈 수 있는데, 나중에 더 빠른 경로를 통해서 같은 지점을 지나가면 return 하면 안 되고, 거리를 더 작은 거리로 업데이트하고 계속 진행해야 하기 때문이다. 따라서, BFS 처럼 단순히 방문 여부만 체크하는 것이 아닌, 거리까지 저장해주고 있어야 한다. 초기화는 매우 큰 정수 값인 inf 값으로 해주었고, 이는 즉 방문하지 않았음을 뜻한다. 이 처리를 통과하면 BFS 와 같은 방식으로 경로 저장을 위해서 prev[nx][ny] 에 (x,y)를 저장해주고, visited[nx][ny] = visited[x][y] + 1 을 해준다. 그리고 found = dls((nx,ny), limit, k+1) 을 통해서 현재 깊이를 1 증가시켜 재귀적으로 DFS 연산을 한다. Dls 의 반환값은 목표지점을 찾았는지의 여부 True or False 이다. DLS 는 이런 방식으로 작동되고, ids 함수 안에서는 while True, 즉 목표지점을 찾을 때까지 depth 를 1 씩 증가시키면서 dls 를 호출한다. 매번 visited 리스트를 Inf 로

초기화해준다. 만약 found 가 True 라면 목표 지점을 찾았다는 뜻이므로, BFS 때와 동일하게 경로를 path 리스트에 저장한다.

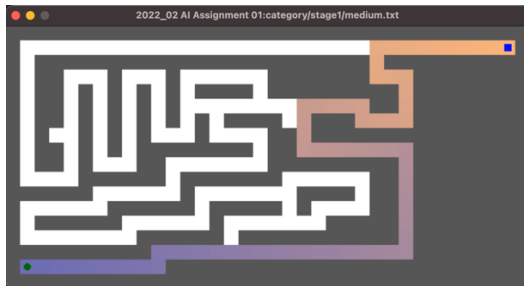
```
# astar
```

Astar / Small



```
=====
[ astar results ]
(1) Path Length: 9
(2) Search States: 14
(3) Execute Time: 0.0000729561 seconds
=====
```

Astar / Medium



```
=====
[ astar results ]
(1) Path Length: 69
(2) Search States: 182
(3) Execute Time: 0.0004248619 seconds
=====
```

Astar / Big

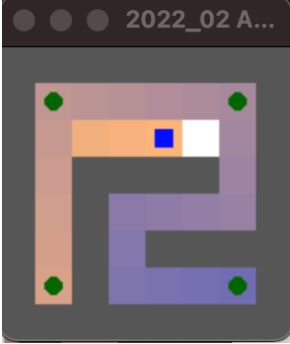


Stage1 의 astar 알고리즘의 heuristic function 은 현재 지점으로부터 도착지까지의 맨해튼 거리로 정의되어 있다. 다익스트라 알고리즘의 기본적인 골격은 그대로 유지하지만, 차이점은 우선순위 큐에 들어가야 할 값이 $F(n)$ 의 값으로, $F(n) = g(n) + h(n)$ 으로 정의된다. 여기서, $g(n)$ 은 출발지점부터 현재 지점까지의 실제 거리를 나타내고, $h(n)$ 은 현재 위치부터 도착지점까지의 맨해튼 거리가 될 것이다. 휴리스틱 함수로 쓰인 맨해튼 거리는 consistent 하다는 조건을 만족하기 때문에 곧 admissible 하고, graph search 를 하는 astar 알고리즘은 optimal solution 을 보장한다. 맨해튼 거리가 consistent 한 이유는 현재 지점 n 으로부터 도착지까지의 맨해튼 거리는 현재 지점으로부터 이동 가능한 인접한 지점 n' 까지의 거리(인접한 칸이기 때문에 1) + n' 으로부터 도착지까지의 맨해튼 거리와 같기 때문이다. 즉, $h(n) == c(n, n') + h(n')$ 이다.

Stage 2.

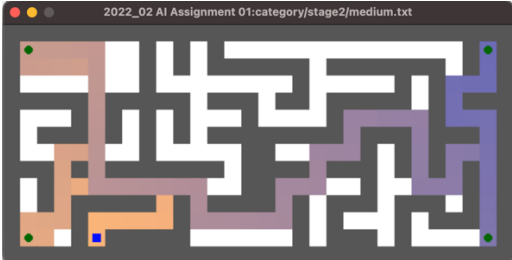
astar_four_circles

Astar_four_circles / Small



```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_four_circles results ]
(1) Path Length: 29
(2) Search States: 38
(3) Execute Time: 0.0008809566 seconds
=====
```

Astar_four_circles / Medium



```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_four_circles results ]
(1) Path Length: 107
(2) Search States: 324
(3) Execute Time: 0.0051589012 seconds
=====
```

Astar four circles / Big



```
class Node(object):
    def __init__(self, past, cur):
        self._past = past # 이전 노드의 위치 튜플
        self._coordinate = cur # 현재 노드의 위치 튜플
        self._leftBit = 0 # 현재 노드의 남은 도착지점들에 대한 비트 정보, 후에 초기화 된다. 해당 인덱스 비트가 1이면 아직 방문하지 않았다는 뜻.
        self._f = 0 # 현재 노드의 f 값
        self._g = 0 # 현재 노드의 g 값
        self._h = 0 # 현재 노드의 휴리스틱 함수 값

    def __lt__(self, other): # 노드를 우선순위 큐에서 F 값을 기준으로 오름차순 정렬하기 때문에 매직 메소드 lt 정의
        return self._f < other._f
```

Astar_four_circles 부터는 각 칸의 정보를 저장하기 위해 Node 객체를 따로 만들어주었다. Stage 1 의 astar 처럼 다음으로 향해야 할 도착지의 정보가 명확하지 않고, 4 개의 도착지점을 방문하는 순서별로 경로의 길이는 달라질 수 있기 때문에 단순히 (x, y) 를 방문했다고 하여 dist[x][y] 처럼 방문처리를 하고, 다시 방문하지 않도록 하면 안 된다. 예를 들어, 도착지 1, 도착지 2, 도착지 3, 도착지 4 가 있는데, 도착지 4 를 먼저 방문했다고 하여 dist[도착지 4 의 x 좌표][도착지 4 의 y 좌표] 를 방문 체크하고 다시는 방문하지 않도록 하면, 다른 순서로 방문하는 경우의 수를 고려하지 않게 된다. 만약 최단 경로가 도착지 1 부터 먼저 방문하는 것이었다면, 최단 경로를 찾지 못하게 된다. 단순히 각 칸의 좌표 값만을 활용했던 stage 1 과는 달리, 조금 더 자세한 노드의 정보를 갖는 노드 객체의 정의가 필요했다. 이 노드 객체는 astar_many_circles 에도 쓰인다. 각 인스턴스 변수에 대한 설명은 위 사진의 주석을 참고하면 되겠다.


```

def stage2_heuristic(N, Cur, index, to_coord, edge_dist):
    if Cur._leftBit == 0: # 남은 목표 지점이 없으면 휴리스틱 함수의 반환 값은 0이다.
        return 0

    left = []

    for i in range(N):
        if (1 << i) & Cur._leftBit: # 해당 인덱스의 bit and 연산이 양수면, 아직 방문하지 않은 목표 지점이란 뜻이므로 left에 추가
            left.append(i)

    permu = itertools.permutations(left) # 남은 목표 지점들의 순열을 모두 구한다.

    # stage 2에서 순열을 이용한 이유는 도착지점이 4개로 고정되어 있고, 4! == 24 밖에 되지 않기 때문이다.
    # 즉, Big-0로 따졌을 때도 무시할 수 있을만한, 그리 크지 않은 상수 값이다.
    cur_x, cur_y = Cur._coordinate

    Min = math.inf
    for p in permu:
        total_dist = 0
        for i in range(len(p)):
            if i == 0: # 현재 지점에서부터 순열의 첫 번째 지점까지는 맨해튼 거리로 구해서 더해준다.
                total_dist += (abs(cur_x - to_coord[p[i]][0]) + abs(cur_y - to_coord[p[i]][1]))
            else: # 나머지는 미리 구해둔 도착점과 도착점 사이의 실제 거리를 더해준다.
                total_dist += edge_dist[p[i]][p[i-1]]

        Min = min(Min, total_dist) # 모든 순열 경우의 수에서 남은 거리가 가장 작은 것을 반환
    return Min

```

Stage2 의 휴리스틱 함수는 현재 노드에서 아직 방문하지 않은 도착지점들의 permutations 를 구한 다음, 각각의 방문 순서 순열에 대해서 $H(n)$ 은 현재 노드부터 해당 순열의 첫번째 도착점까지의 맨해튼 거리 + 첫번째 도착점부터 남은 도착점들 간 거리는 미리 계산해둔 실제 거리(edge dist)를 활용하여 더해주었다. 모든 순열의 경우의 수에 대해서 가장 작은 값을 휴리스틱 함수의 return 값으로 반환해주었다. 이 휴리스틱 함수는 consistent 하다. 그 이유는 현재 지점을 n 이라고 할 때, 현재 지점에서의 $h(n)$ 값은 현재 지점으로부터 인접한 노드 n' 까지의 거리 + $h(n')$ 의 값과 같다. 즉, $h(n) = c(n, a, n') + h(n')$ 을 만족한다. 왜냐하면 $c(n, a, n')$ 은 인접한 칸이므로 1 이고, $h(n) = 1 + h(n')$ 인데, 최적의 방문 순서가 동일하면, 결국 첫번째 도착점부터 남은 도착점들 간 거리는 n 과 n' 이 같고, 결국 다른 것은 n 부터 첫번째 도착점까지의 거리, 그리고 n' 부터 첫번째 도착점까지의 거리인데, n 부터 첫번째 도착점까지의 거리를 D 라고 하면, n' 부터 첫번째 도착점까지의 거리는 $D - 1$ 이다. 현재 지점부터 첫번째 도착점까지의 거리는 맨해튼 거리로 계산했기 때문이다. 따라서 $h(n) = D +$ 첫번째 도착점부터 남은 도착점들 간 거리의 합, $h(n') = D - 1 +$ 첫번째 도착점부터 남은 도착점들 간 거리의 합이므로, 이를 원래의 식에 대입하여 전개하면 $h(n) - c(n, a, n') - h(n') = D +$ 첫번째 도착점부터 남은 도착점들 간 거리의 합 $- 1 - D + 1 -$ 첫번째 도착점부터 남은 도착점들 간 거리의 합이므로 0 이 되면서 원래의 가정을 만족한다. Consistent 하다는 조건이 만족되었으므로, admissible 하고, graph search 를 이용한 astar 알고리즘에서 optimal solution 을 만족한다. 이 휴리스틱이 admissible 한 것은 자명한데, 첫번째 도착점까지는 실제 거리를 사용하지 않고 맨해튼 거리를 사용하기 때문에 실제 거리보다 작거나 같을 수 밖에 없다.

위에서 설명한 도착지점들 간 실제거리는 BFS 알고리즘을 통해서 미리 전처리를 해주고, stage2_heuristic 함수의 인자 edge_dist 로 넘겨주었다.

```

for cx, cy in Start._leftCircles: # 0-1 BFS 알고리즘을 이용해서 미리 도착 지점들 사이의 실제 거리를 구한다.
    # 미로의 각 인접한 칸은 애지가 1인 그래프로 생각해도 되기 때문에 굳이 다익스트라를 통한 최단 거리를 구할 필요가 없다.
    # BFS를 돌면서 특정 지점을 처음 만난 경우에 그때의 길이가 해당 지점과의 최단 거리다.
    for i in range(n+1):
        for j in range(m+1):
            dist[i][j] = math.inf

    dist[cx][cy] = 0
    queue = deque([(cx,cy)])
    while queue:
        x, y = queue.popleft()

        if (x != cx or y != cy) and maze.isObjective(x,y) == True:
            edge_dist[coord_to_idx[(cx,cy)]] [coord_to_idx[(x,y)]] = dist[x][y]
            edge_dist[coord_to_idx[(x,y)]] [coord_to_idx[(cx,cy)]] = dist[x][y]

        for i in range(4):
            nx = x + dx[i]
            ny = y + dy[i]

# neighborPoints를 쓰지 않은 이유는 전처리 과정은 실제 에이스타 과정에서 노드를 Expand 한 것이 아니기 때문에 Search States에 포함하지 않았다.
        if nx < 0 or nx >= n or ny < 0 or ny >= m or maze.isWall(nx, ny):
            continue

        if dist[x][y] + 1 < dist[nx][ny]:
            dist[nx][ny] = dist[x][y] + 1
            queue.append((nx,ny))

```

BFS 를 통해서 각 도착지점들 간 최단 거리를 찾을 수 있는 이유는 미로는 각 인접한 칸으로까지의 거리가 1로 고정되어 있기 때문에 BFS 를 하면서 최단 경로를 찾아줄 수 있기 때문이다. 즉, 각 도착점에서 대해서 0-1 BFS 를 하면서 이를 저장해준다.

```

while pq:
    d_v, Cur = heapq.heappop(pq)
    x, y = Cur._coordinate

    if d_v != Cur._f: # heappush를 할 때의 거리가 현재 저장되어 있는 F[x][y]와 다르다면, F[x][y]가 작아졌다는 뜻이므로 불필요없이 continue
        continue

    if Cur._leftBit == 0: # 모든 도착지를 방문했으면
        cur_x, cur_y = x, y
        state = 0 # 모든 도착지를 방문한 상태는 0이다. 처음 시작 상태가 모두 1로 되어있는 상태였으니.
        while (cur_x != start_point[0] or cur_y != start_point[1]) or (state != ((1 << len_circles) - 1)): # 시작 좌표, 시작 상태까지
            path.append((cur_x, cur_y)) # 경로 추가
            ((cur_x, cur_y), state) = pre(((cur_x, cur_y), state)) # 경로 정보를 불러와서 현재로 업데이트
            path.append(start_point)
            path.reverse()
            break

    nxt = maze.neighborPoints(x, y)

    for nx, ny in nxt:
        next_node = Node(Cur, (nx,ny)) # 다음 노드 생성
        next_node._leftBit = Cur._leftBit # 다음 노드의 남은 도착지 비트 정보를 현재 노드의 도착지 비트 정보로 업데이트

# 만약 다음 노드가 도착지이고, 아직 방문하지 않은 도착지인 경우에는 통글하여 해당 비트를 0으로 바꾸어 주어야 한다.
        if goal_list[(nx,ny)] == 1 and ((1 << coord_to_idx[(nx,ny)]) & Cur._leftBit):
            next_node._leftBit ^= (1 << coord_to_idx[(nx,ny)]) # 해당 도착지 방문 표시를 해준다. 해당 노드의 인덱스 번호 비트 1 -> 0

        if closed[(Cur._coordinate, Cur._leftBit)] + 1 < closed[(next_node._coordinate, next_node._leftBit)]: # astar 에서 설명한 것과 동일 현재까지 거리 + 1 < 다음 노드까지의 거리일 때만 expand
            next_node._g = closed[(Cur._coordinate, Cur._leftBit)] + 1 # 출발지부터 다음 노드까지의 거리 = 출발지부터 현재 노드까지의 거리 + 1
            next_node._h = stage2_heuristic(len_circles, next_node, coord_to_idx, idx_to_coord, edge_dist)
            next_node._f = next_node._g + next_node._h
            closed[(next_node._coordinate, next_node._leftBit)] = next_node._g # 방문 표시 업데이트
            pre[(next_node._coordinate, next_node._leftBit)] = (Cur._coordinate, Cur._leftBit) # 경로 역추적을 위한 업데이트
            heapq.heappush(pq, (next_node._f, next_node))

return path

```

먼저 시작 지점의 휴리스틱 값을 우선순위 큐에 넣어주고 시작한다. 우선순위 큐에 들어가는 자료구조는 (해당 노드의 F 값, 해당 노드 객체) 로서, 노드의 F 값을 기준으로 정렬이 되어 이 값이 작은 순서대로 우선순위 큐에서 노드가 선택될 것이다. 여기서 눈여겨 보아야 할 것은 바로 closed 변수이다. 이는 방문 처리를 위한 defaultdict 로서, (노드의 좌표, 노드의 남은 목표

지점들에 대한 비트 정보) 와 같은 튜플 형식의 키를 갖고, 값으로는 출발 지점부터 현재 노드까지의 실제 거리를 저장한다. (g 값) 단순 좌표로 방문 처리를 하지 못하기 때문에 좌표에다가 남은 도착점들의 정보를 추가함으로써 단순히 해당 좌표에 방문했다고 해서 다시 방문하지 못하게 하는 것이 아니라 모든 순열에 대해서 해당 지점을 다시 방문할 수 있도록 해주었다. 하지만 같은 방문 순서의 노드 같은 경우에 대한 중복 처리를 방지하기 위해 위와 같은 형식으로 방문 체크를 하게 된 것이다.

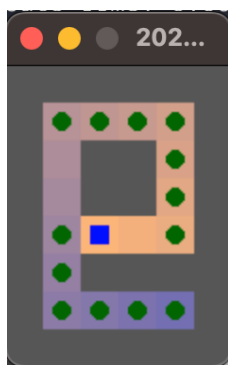
그 외 에이스타 알고리즘의 동작 방식은 stage1 과 비슷하나, 달라진 점은 좌표에 대해서 연산을 하는 것이 아닌 노드에 대해서 한다는 점이다. 먼저 현재 지점에서 남은 도착점들에 대한 비트 정보가 0 이면 다 방문하였다는 뜻이므로, 탈출 조건이 된다. 그리고 현재 지점으로부터 neighborPoints 메소드를 통해서 다음 갈 수 있는 지점들의 후보를 가져와서 각각의 좌표에 대해서 이제 g 값과 h 값 f 값을 업데이트해준다. 만약 다음 후보가 남은 도착점들 중 하나라면, 해당 도착에 대한 비트를 현재 도착점들 비트에서 토글하여 0 으로 바꾸어준다. 또한, 나중에 도착지점을 모두 방문하고 경로의 역추적을 위해서 closed 와 같은 형식의 키값을 가지는 pre default dict 을 선언해주었고, 이는 값으로 (이전 노드의 좌표, 이전 노드의 남은 도착점들에 대한 비트 정보)와 같이 저장되어 있다.

만약 모든 도착점들을 방문했다면, 비트가 0 일 것이고, 경로 역추적을 해나가면 되는데, 이전과 달라진 점은 이전에는 바로 좌표 값만 역으로 업데이트해나갔지만, 여기서는 비트 정보도 이전 노드의 비트 정보로 업데이트 한다는 것이다. 당연한 것이 (현재 노드의 좌표, 현재 노드의 남은 도착점들에 대한 비트 정보) 로 방문 처리를 했기 때문에, 역추적할 때도 이 정보를 이용해야 한다.

Stage 3.

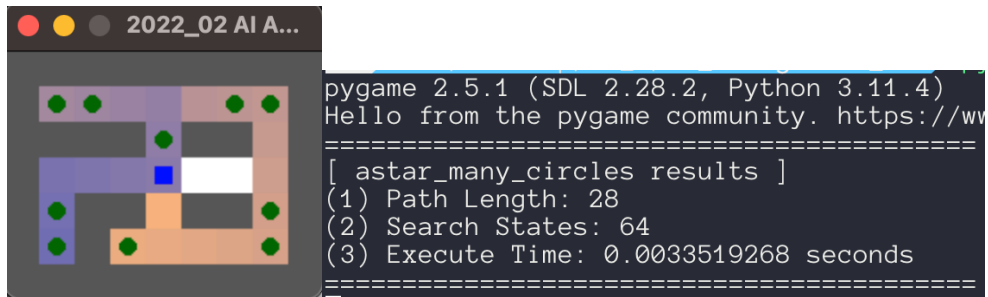
astar_many_circles

Astar_many_circles / tiny



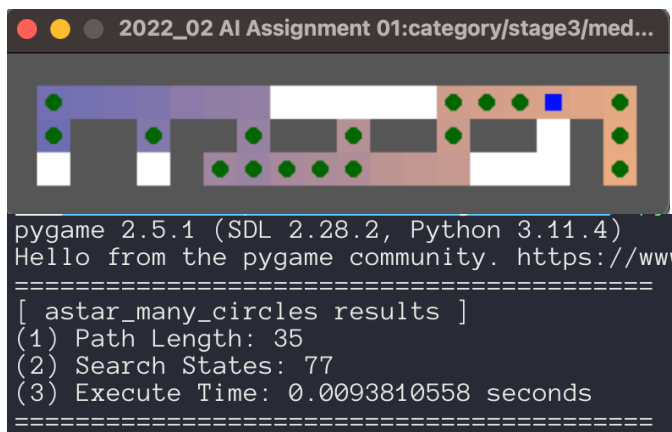
```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_many_circles results ]
(1) Path Length: 17
(2) Search States: 17
(3) Execute Time: 0.0021498203 seconds
=====
```

Astar many circles / Small



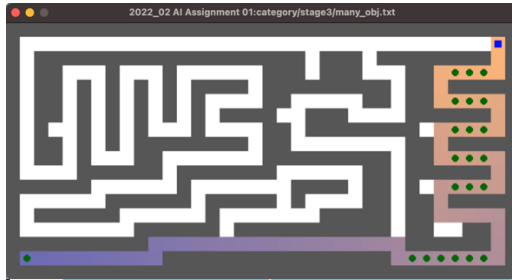
```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_many_circles results ]
(1) Path Length: 28
(2) Search States: 64
(3) Execute Time: 0.0033519268 seconds
=====
```

Astar many circles / Medium



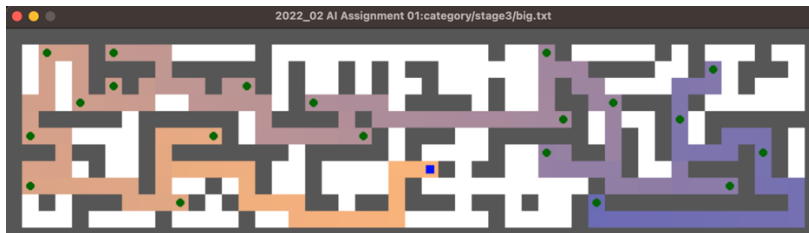
```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_many_circles results ]
(1) Path Length: 35
(2) Search States: 77
(3) Execute Time: 0.0093810558 seconds
=====
```

Astar_many_circles / Many_obj



```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_many_circles results ]
(1) Path Length: 75
(2) Search States: 91
(3) Execute Time: 0.0175302029 seconds
=====
```

Astar_many_circles / Big



```
pygame 2.5.1 (SDL 2.28.2, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org
=====
[ astar_many_circles results ]
(1) Path Length: 199
(2) Search States: 38511
(3) Execute Time: 5.0810260773 seconds
=====
```

```

def stage3_heuristic(N, Cur, index, to_coord, adj):
    if Cur._leftBit == 0:
        return 0

    ## 현재 노드에서 남은 도착지점들까지의 맨해튼 거리를 구하고, 그 중 가장 작은 거리를 ndist에 저장한다.
    cx, cy = Cur._coordinate
    nx, ny = None, None
    Min = math.inf
    for i in range(N):
        if ((1 << i) & Cur._leftBit): # 해당 인덱스가 아직 방문해야 하는 도착지라면
            if abs(to_coord[i][0] - cx) + abs(to_coord[i][1] - cy) < Min: # 현재 지점부터 해당 도착지까지의 맨해튼 거리 구하기
                Min = abs(to_coord[i][0] - cx) + abs(to_coord[i][1] - cy) # 그 중에서 가장 Min 값을 취할 것임.
                nx, ny = to_coord[i] # 해당 도착지의 좌표 저장

    ndist = Min # 선택된 다음 목적지까지의 맨해튼 거리

    # 프림 알고리즘을 사용한 MST 생성 / complete graph 이므로  $O(V^2)$ 에 동작
    dist = [math.inf] * (N+1) # 현재 Tree에 속한 노드들과 해당 인덱스까지의 최소 거리
    visited = [0] * (N+1) # 이미 트리에 속한 노드인지
    dist[index[(nx,ny)]] = 0 # 처음 도착지는 0으로 시작
    MST_WEIGHT = 0 # MST 의 총 비용

    for i in range(N):
        now = -1
        min_dist = math.inf
        if (1 << i) & Cur._leftBit == 0: # 이미 방문한 도착지는 MST에 포함시키지 않아야 한다.
            continue
        for j in range(N):
            if (1 << j) & Cur._leftBit == 0: # 이미 방문한 도착지는 MST에 포함시키지 않아야 한다.
                continue
            if visited[j] == 0 and dist[j] < min_dist: # 트리에 추가할 다음 노드 후보 정하기
                min_dist = dist[j]
                now = j
        if now < 0:
            continue
        MST_WEIGHT += min_dist # 트리에 해당 노드까지의 거리 추가
        visited[now] = 1 # 방문 표시

        for j in range(N):
            if (1 << j) & Cur._leftBit == 0: # 이미 방문한 도착지는 MST에 포함시키지 않아야 한다.
                continue
            dist[j] = min(dist[j], adj[now][j]) # 트리의 추가된 노드로부터 dist 갱신

    return ndist + MST_WEIGHT # 반환 값은 (현재 위치 ~ 처음 도착지까지의 맨해튼 거리) + 방문하지 않은 도착지들 간의 MST 비용

```

Stage3의 휴리스틱 함수는 stage2의 휴리스틱 함수와 많이 다르다. Stage3에서는 순열을 사용할 수 없는 것이 도착지점들의 개수가 많기 때문에 시간복잡도가 기하급수적으로 커지게 된다. 그래서 고안한 방법이 MST를 이용한 방법이다. 먼저 인자로 도착지점들의 개수, 현재 노드 객체, 좌표를 인덱스로 변환해주는 index 딕셔너리, 인덱스를 다시 좌표로 변환해주는 to_coord 딕셔너리, 그리고 도착지점들 간의 실제 최단 거리가 저장되어 있는 adj 인접 행렬이 있다. 휴리스틱 함수의 반환 값은 (현재 노드부터 아직 방문하지 않은 도착점들 중에서 맨해튼 거리가 가장 짧은 거리) + (아직 방문하지 않은 도착점들로 만든 MST의 비용)이다. 현재 노드의 도착점들에 대한 비트 정보를 활용해서 아직 방문하지 않은 도착점이라면 맨해튼 거리를 각각 구하고, 가장 가까운 도착점을 찾고 해당 도착점까지의 거리를 ndist에 저장해준다. 그리고 프림 알고리즘을 통해서 아직 방문하지 않은 도착점들 간의 MST를 만들며 비용을 구한다.

프림 알고리즘의 간단한 동작 방식은 다음과 같다. 크루스칼 알고리즘이 간선 비용이 작은 것부터 골라 나가는 것과 달리, 프림 알고리즘은 시작 노드를 선택하고, 그 노드와 인접한

노드까지의 거리 중에서 트리에 추가할 노드를 거리가 짧은 노드를 기준으로 넓혀나간다. 즉, 시작 노드부터 하나의 트리를 MST로 만들어 나가는 것이다. (크루스칼은 여기 저기서 트리가 만들어지고, 나중에 하나로 합쳐지는 느낌이다.) 다음 노드가 선택되었다면, 해당 노드는 시작 노드와 연결되었고, 그 트리부터 다음 후보까지의 거리를 갱신한다. 그리고 위 과정을 MST가 만들어질 때까지 반복한다. (총 N번, 한번에 하나씩 expand 해나가기 때문이다) 하지만, 나는 아직 방문하지 않은 도착점들에 대해서만 MST를 만들고 싶기 때문에 해당 인덱스와 현재 노드의 비트를 and 연산한 값이 0이면, 이미 방문한 도착점이라는 뜻이므로 continue 해주었다.

이 휴리스틱 함수는 consistent 하고, 즉 admissible 하다. $h(n) \leq c(n, a, n') + h(n')$ 을 만족하는데, 현재 위치가 n 이라고 가정해보면, $h(n)$ 은 현재 위치부터 방문하지 않은 도착점 중 가장 가까운 도착점까지의 맨해튼 거리 + 남은 도착점 간의 MST 비용이다. $h(n')$ 은 n' 으로부터 방문하지 않은 도착점 중 가장 가까운 도착점까지의 맨해튼 거리 + 남은 도착점 간의 MST 비용이다. n 과 n' 은 인접하므로 방문하지 않은 도착점 중 가장 가까운 도착점까지의 맨해튼 거리는 1 차이가 난다. 하지만 MST 비용 값은 동일하다. 따라서, $h(n) == c(n, a, n') + h(n')$ 을 만족하게 돼 consistent 하며, admissible 하게 된다.

```
while pq:
    d_v, Cur = heapq.heappop(pq)
    x, y = Cur._coordinate

    if d_v != Cur._f: # heappush를 할 때의 거리가 현재 저장되어 있는 F[x][y]와 다르면, F[x][y]가 작아졌다는 뜻이므로 불필요없이 continue
        continue

    if Cur._leftBit == 0:
        cur_x, cur_y, cur_g = x, y, Cur._g
        state = 0 # 모든 도착지를 방문한 상태는 0이다. 처음 시작 상태가 모두 1로 되어있는 상태였으니.
        while (cur_x != start_point[0] or cur_y != start_point[1]) or (state != ((1 << len_circles) - 1)): # 시작 좌표, 시작 상태까지
            path.append((cur_x, cur_y))
            ((cur_x, cur_y), state) = pre[((cur_x, cur_y), state)]
            cur_g -= 1
        path.append(start_point)
        path.reverse()
        break

    nxt = maze.neighborPoints(x, y)

    for nx, ny in nxt:
        next_node = Node(Cur, (nx, ny))
        next_node._leftBit = Cur._leftBit

# 만약 다음 노드가 도착지이고, 아직 방문하지 않은 도착지인 경우에는 토글하여 해당 비트를 0으로 바꾸어 주어야 한다.
if goal_list[(nx, ny)] == 1 and ((1 << coord_to_idx[(nx, ny)]) & Cur._leftBit):
    next_node._leftBit ^= (1 << coord_to_idx[(nx, ny)]) # 해당 도착지 방문 표시를 해준다. 해당 노드의 인덱스 번호 비트 1 -> 0

if closed[(Cur._coordinate, Cur._leftBit)] + 1 < closed[(next_node._coordinate, next_node._leftBit)]:
    next_node._g = closed[(Cur._coordinate, Cur._leftBit)] + 1
    next_node._h = stage3_heuristic(len_circles, next_node, coord_to_idx, idx_to_coord, adj) # 이 부분만 stage2와 다른 부분이다.
    next_node._f = next_node._g + next_node._h
    closed[(next_node._coordinate, next_node._leftBit)] = next_node._g
    pre[(next_node._coordinate, next_node._leftBit)] = (Cur._coordinate, Cur._leftBit)
    heapq.heappush(pq, (next_node._f, next_node))

return path
```

Stage3의 astar_many_circles 함수는 stage 2의 astar_four_circles와 거의 동일한 수준이고, 이에 대한 설명은 위에 자세히 했으므로 여기서는 생략한다. 다른 코드는 휴리스틱 호출 부분밖에 없다.