자율 청소 로봇 프로젝트

B-5 조

팀원 : 김주원/ 류재준 / 양준혁 /이상우

목차

Miri Company

프로젝트 주제 및 목표

- 프로젝트 주제
  - 자동 Mapping 및 청소, visual tracking 기능을 갖춘 로봇 청소기 개발

- 프로젝트 목표
  - 자율적인 맵 작성
    - 로봇이 환경을 탐색하며 실시간으로 맵을 생성
  - 자동 청소 수행
    - 맵이 완전히 확보되면, 로봇은 최적의 청소 경로를 따라 자동으로 청소 작업을 수행
  - Visual tracking 기능 통합
    - ORB 알고리즘과 BFMatcher를 활용하여 로봇에 장착된 카메라로 주변을 실시간으로 관찰

팀원 소개

김주원	자율 탐색 알고리즘 개발 및 튜닝 / 청소 알고리즘 개발
류재준	청소 알고리즘 개발 / 자율 탐색 튜닝 전략 검토
양준혁	Visual Tracking 알고리즘 개발 / 자율 탐색 알고리즘 개발 및 튜닝
이상우	청소 알고리즘 개발

자율 탐색 알고리즘

알고리즘 Work flow

- 1) Frontier 탐색
  - Frontier: 탐색되지 않은 영역과 탐색된 영역의 경계
  - 목적: 로봇이 새로운 영역을 탐색하도록 유도
- 2) 목표 선택 및 발행
  - Frontier 감지: 맵에서 Frontier 영역을 탐지
  - 벽 근처 Frontier 필터링: 장애물과 너무 가까운 Frontier는 제외
  - 목표 선택: 유효한 Frontier 중 하나를 랜덤으로 선택
  - 목표 발행: 선택된 Frontier를 목표로 설정하여 로봇 이동 유도
- 3) 목표 상태 관리
  - 목표 상태 확인: 목표가 성공적으로 완료되었는지, 실패했는지 확인
  - 목표 재설정: 목표가 완료되었을 경우, 새로운 Frontier 탐색 시작
- 4) 종료 확인
  - 맵을 모두 탐색하고, 더 이상 Frontier를 찾을 수 없다면 자율 탐색 종료

자율 탐색 알고리즘 pseudo code

```
BEGIN FrontierExplorationNode

    INITIALIZE Node as 'frontier_exploration_node'

    SUBSCRIBE to 'map' with callback map_callback
    SUBSCRIBE to '/follow_path/_action/status' with callback goal_status_callback
    SUBSCRIBE to '/odom' with callback odom_callback
    PUBLISH to 'goal_pose'

    INITIALIZE variables:
        map_array = None
        map_metadata = None
        goal_reached = True
        robot_x = 0.0
        robot_y = 0.0

    CREATE timer to call timer callback every 5 seconds

    FUNCTION map_callback(msg):
        CONVERT OccupancyGrid data to numpy array
        RESHAPE to (height, width)
        STORE in map_array
        STORE map info in map_metadata

    FUNCTION goal_status_callback(msg):
        IF there are statuses in msg:
            GET the latest status
            IF status is SUCCEEDED or ABORTED or REJECTED or FAILED:
                SET goal_reached to True
                LOG corresponding message
            ELSE:
                SET goal_reached to False
                LOG current status

    FUNCTION odom_callback(msg):
        UPDATE robot_x and robot_y with odometry data
```

1) Subscriber 설정:
맵 데이터 (map): map 토픽을 subscribe하여 맵 정보 업데이트
목표 상태 (/follow_path/_action/status): 현재 목표의 상태를 subscribe하여 목표 완료 여부 확인
오도메트리 데이터 (/odom): 로봇의 현재 위치와 자세 정보를 subscribe하여 로봇의 odom 상태를 추적

2) Publisher 설정:
목표 포즈 (goal_pose): 로봇이 이동할 목표 위치를 publish

3) 변수 초기화:
map_array: 맵 데이터를 저장할 numpy 배열. 초기값은 None.
예시 : -1은 알 수 없는 영역 (Unknown), 0은 자유 공간 (Free), 100은 장애물 (Occupied)
map_metadata: 맵의 메타데이터를 저장. 초기값은 None.
goal_reached: 목표 완료 여부를 추적. 초기값은 True로 설정하여, 시작 시 바로 Frontier 탐색을 시도.
robot_x, robot_y: 로봇의 현재 위치를 저장. 초기값은 0.0.

4)타이머 설정:
5초마다 timer_callback 함수를 호출하여 Frontier 탐색을 주기적으로 수행

5) 콜백 함수 설명:
map_callback: 수신한 OccupancyGrid 형식의 데이터를 numpy 배열로 변환하고, 맵의 메타데이터를 저장
goal_status_callback: 목표 상태를 확인하고, 목표가 완료되었는지 여부를 업데이트함 / 성공, 취소, 실패 등의 상태를 로그로 출력
odom_callback: 오도메트리 데이터를 통해 로봇의 현재 위치를 업데이트

자율 탐색 알고리즘 pseudo code

```
FUNCTION timer_callback():
    IF map_array or map_metadata is None:
        RETURN

    IF not goal_reached:
        LOG "목표 지점에 도착하지 않았습니다. 프론티어 탐색을 skip합니다."
        RETURN

    DETECT frontiers using detect_frontiers()

    IF no frontiers detected:
        LOG "프론티어를 찾을 수 없습니다."
        RETURN

    FILTER frontiers to exclude those near walls using is_near_wall()

    IF no valid frontiers left:
        LOG "모든 프론티어가 벽(장애물) 근처에 존재합니다. 유효한 프론티어가 존재하지 않습니다."
        RETURN

    SELECT a random goal from valid frontiers using select_goal()

    IF goal selection failed:
        LOG "목표지점을 찾는데 실패하였습니다."
        RETURN

    PUBLISH the selected goal using publish_goal(goal_xy)
```

1) 타이머 콜백 (timer_callback):

맵 데이터 확인:

- map_array와 map_metadata가 None인지 확인 => 맵 데이터가 아직 수신되지 않았다면, 탐색을 건너뜀

2) 목표 상태 확인:

- 현재 로봇이 목표를 향해 이동 중인지 (goal_reached=False) 확인
- 목표 지점에 아직 도달하지 않았다면, 새로운 Frontier 탐색을 건너뜀 => 이는 목표 수행 중 간섭을 방지함

3) Frontier 감지 (detect_frontiers):

맵에서 Frontier 영역 감지

- Frontier란 탐색되지 않은 영역과 탐색된 영역의 경계를 의미
- Frontier를 감지함으로써 로봇은 새로운 탐색 영역을 찾아 이동함

Frontier 필터링 (is_near_wall):

- 감지된 Frontier 중 벽(장애물)과 너무 가까운 Frontier는 제외 => 이는 로봇이 벽에 너무 밀착되지 않고 안전하게 탐색할 수 있도록 도와줌
- 필터링 후에도 유효한 Frontier가 남아있는지 확인
- 유효한 Frontier가 없다면, "모든 프론티어가 벽(장애물) 근처에 존재합니다. 유효한 프론티어가 존재하지 않습니다."라는 메시지를 로그로 출력하고, 탐색을 종료

4) 목표 선택 (select_goal):

- 유효한 Frontier 중 하나를 랜덤으로 선택하여 목표로 설정 => 랜덤 선택을 통해 탐색의 다양성을 확보하고, 균형 잡힌 탐색을 유도

5) 목표 발행 (publish_goal):

- 선택된 Frontier를 목표로 설정하고, 이를 PoseStamped 메시지 형식으로 publish

자율 탐색 알고리즘 pseudo code

```
    FUNCTION detect_frontiers():
        FOR each cell in map_array:
            IF cell is UNKNOWN (-1) and has at least one FREE (0) neighbor:
                ADD cell coordinates to frontiers
        RETURN frontiers

    FUNCTION is_near_wall(col, row, map_data, threshold):
        FOR each cell within threshold distance from (col, row):
            IF cell is OCCUPIED (100):
                RETURN True
        RETURN False

    FUNCTION select_goal(frontiers):
        IF frontiers is empty:
            RETURN None
        RANDOMLY CHOOSE a frontier from frontiers
        CONVERT frontier coordinates to real-world coordinates
        RETURN real_x, real_y

    FUNCTION publish_goal(goal):
        CREATE PoseStamped message
        SET header with current time and 'map' frame
        SET position to goal coordinates
        SET orientation to default (no rotation)
        PUBLISH the goal
        LOG the goal coordinates
        SET goal_reached to False

END FrontierExplorationNode

MAIN:
    INITIALIZE rclpy
    CREATE FrontierExplorationNode
    SPIN node
    SHUTDOWN rclpy

END MAIN
```

1) Frontier 탐지 (detect_frontiers)

맵에서 탐색되지 않은 영역(UNKNOWN)과 탐색된 영역(FREE)의 경계 찾는 method

- 동작 방식: 맵의 모든 셀을 검사하여, UNKNOWN 상태(-1)인 셀 중 하나 이상의 FREE 이웃이 있는 셀을 Frontier로 추가
- 결과: 탐색 가능한 Frontier 리스트를 반환

2) 벽 근처 Frontier 필터링 (is_near_wall)

선택된 Frontier가 장애물(벽)과 너무 가까운지 확인하여 안전한 탐색을 도와주는 method

- 동작 방식: Frontier 주변 지정된 거리(threshold) 내에 OCCUPIED (100) 상태의 셀이 있는지 검사 => 장애물이 가까이 있으면 해당 Frontier를 제외
- 결과: Frontier가 벽과 가까우면 True, 아니면 False를 반환

3) 목표 선택 (select_goal)

유효한 Frontier 중 하나를 목표로 선택

- 동작 방식: Frontier 리스트가 비어있으면 목표 선택 실패(None)를 반환 => 리스트에서 랜덤으로 하나의 Frontier를 선택하고, 그 좌표를 실제 좌표로 변환
- 결과: 선택된 목표의 실제 좌표(real_x, real_y)를 반환

4) 목표 발행 (publish_goal)

선택된 Frontier를 로봇의 이동 목표로 설정하여 발행

- 동작 방식: PoseStamped 메시지를 생성하고, 목표 좌표와 기본 방향을 설정 => 메시지를 goal_pose 토픽으로 발행하고, 목표 좌표를 로그에 기록
- goal_reached를 False로 설정하여 로봇이 현재 목표를 향해 이동 중임을 표시
- 결과: 로봇이 새로운 목표를 향해 이동을 시작

자율 탐색 알고리즘 튜닝전략

튜닝 전략 V1

- resolution : 0.02
- 터틀봇의 전체적인 속도 및 가속도를 줄임 (FollowPath)
- sim_time : 1.7에서 6.0으로 높여 실시간 경로 탐지 성능을 높임
- BaseObstacle.scale : 0.02 에서 0.04로 높여 벽과의 충돌을 방지함
- PathDist.scale : 32.0에서 50.0으로 높여 global 경로를 잘 따르도록 설정함
- local_costmap의 cost_scaling_factor & inflation_radius : 값을 조정해가며 실험한 결과 default 값과 큰 변화가 없는 4.0, 0.3으로 설정함

=〉 느리지만 벽에 부딪히지 않고 자율탐색을 완료

=〉 좁은 경로나 벽 근처에서 동작이 느려지는 issue 가 있음

튜닝전략 V2

- 기존에 제공하는 DWBLocalPlanner 플러그인을 사용하지 않고 "RoatationShimController"라는 플러그인을 사용하여 로봇의 회전을 더욱 부드럽게 함
- resolution : 0.01
- local_costmap 의 cost_scaling_factor & inflation_radius : V1과 동일

=〉 V1 보다 빠르게 벽에 부딪히지 않고 자율탐색 완료

=〉 좁은 경로에서 동작이 느려지는 issue가 있음

```
# DWB parameters
FollowPath:
  plugin: "nav2_rotation_shim_controller::RotationShimController"
  primary_controller: "nav2_regulated_pure_pursuit_controller::RegulatedPurePursuitController"
  angular_dist_threshold: 0.785
  forward_sampling_distance: 0.5
  rotate_to_heading_angular_vel: 3.6
  max_angular_accel: 3.2
  simulate_ahead_time: 1.0
  rotate_to_goal_heading: false

  align_before_move: true
  goal_heading_tolerance: 0.05

  # plugin: "dwb_core::DWBLocalPlanner"
```

자율 청소 알고리즘

```
mark_inflated_obstacles(맵 배열, 로봇 반경, 해상도):
  벽에 해당하는 셀 주변
  유클리드 거리 기준
  로봇 반경만큼 확장
  0 -> -1 로 변경
mark_visited(방문 맵, 행, 열, 로봇 반경, 해상도, 맵 배열):
  방문 맵에 해당하는 셀 주변
  유클리드 거리 기준
  로봇 반경만큼 확장
  0 -> 1 로 변경
world_to_map(world_x, world_y, origin_x, origin_y, 해상도):
  row = int((world_y - origin_y) / 해상도)
  col = int((world_x - origin_x) / 해상도)
  return row, col
map_to_world(map_row, map_col, origin_x, origin_y, 해상도):
  위와 반대
```

```
find_max_point(맵 배열, 현재 행, 현재 열, 방향):
  if 방향 == 'down':
    for r = 현재 행 + 1 ~ H-1:
      if 맵 배열[r, 현재 열] == -1:
        return (r-1, 현재 열) if (r-1) >= 현재_행 else None
    return (H-1, 현재_열)
  다른 방향도 마찬가지...
```

자율 청소 알고리즘

Bresenham_line(행0, 열0, 행1, 열1):
정수 연산만을 사용해 두 점 사이를 잇는 직선을 그리는 알고리즘
임베디드나 실시간 성능이 중요한 경우 사용할 수 있음
선의 기울기 결정
실제 선과 선택한 픽셀 간의 오차를 계산해 다음 픽셀 방향 결정
오차를 기반으로 현재 픽셀에서 다음 픽셀 결정
d_row, d_col = 두 점 사이의 행과 열의 차이
s_row, s_col = 행과 열의 증감 방향
err = 오차(초기값 d_col - d_row)
오차에 따라 증감

extract_region(맵 배열, 시작 행, 시작 열):
시작 위치를 기준으로 연결된 0을 추출, 나머지를 -1
BFS(Breadth-First Search) 기반의 Flood Fill 알고리즘 사용
BFS : 너비 우선 탐색 - 시작 노드로부터 인접한 노드를 먼저 탐색하고, 그 다음 레벨의 노드를 순찾거으로 탐색
Flood Fill : 시작 지점과 인접한 셀이 동일한 값인지를 재귀적으로 탐색
시작 위치를 큐에 추가
상하좌우 네 방향으로 큐가 빌 때까지 0을 탐색
각각의 region map이 추출됨

자율 청소 알고리즘

```
coverage_pass(현재 행, 현재 열, 방향 우선순위, origin_x, origin_y, 해상도,
              확장된 맵, 방문한 맵, 커버리지 경로, 목표 지점들, 로봇 반경, 맵 배열):
 while True:
  find_max_point(확장된 맵, 현재 행, 현재 열, 방향 우선순위)
  if 목표 지점 == 없음 or 목표 지점 == -1(안전구역) or 목표 지점 == 1(방문):
    break
    목표 지점 변환
     경로 = Bresenham_line(현재 행, 현재 열, 목표 행, 목표 열)
  for 각 셀 in 경로
    경로에 추가
    방문 표시
 return 경로
```

```
generate_coverage_path(맵 배열, 확장된 맵, start_x, start_y, 로봇 반경, 해상도,
origin_x, origin_y):
  왼쪽 위 코너 find_max_point(왼쪽, 위)
    커버리지 경로 확장
  아래 find_max_point(아래)
    커버리지 경로 확장
  오른쪽 move_right(로봇 반경)
  위 find_max_point(아래)
    커버리지 경로 확장
  방문하지 않은 셀 찾기
   if 방문하지 않은 셀 == None
     break
```

Visual Tracking 알고리즘

```
BEGIN VisualTrackingNode

    INITIALIZE Node as 'visual_tracking_node'

    # ---------------------------- 이미지 구독 설정 ----------------------------
    SUBSCRIBE to '/oakd/rgb/preview/image_raw/compressed' with callback image_callback

    # ---------------------------- CvBridge 초기화 ----------------------------
    INITIALIZE CvBridge for ROS <-> OpenCV 이미지 변환

    # ---------------------------- ORB 설정 ----------------------------
    CREATE ORB detector with nfeatures=30
    LOG "ORB 생성 완료."

    # ---------------------------- BFMatcher 설정 ----------------------------
    CREATE BFMatcher with NORM HAMMING
    LOG "BFMatcher를 사용합니다."

    # ---------------------------- 이전 프레임 저장용 ----------------------------
    SET prev_keypoints and prev_descriptors to None

    # ---------------------------- 시각화용 설정 ----------------------------
    SET display_width to 640
    SET display_height to 480
    CREATE OpenCV window named 'Visual Tracking - ORB' with size 640x480

    # ---------------------------- 잔상(페이드) 효과 설정 ----------------------------
    INITIALIZE overlay image with zeros
    SET fade_factor to 0.9
    SET overlay_alpha to 0.7

END VisualTrackingNode
```

```
FUNCTION image_callback(msg):
    TRY:
        CONVERT compressed_image msg to OpenCV image (cv_image)
    EXCEPT:
        LOG "이미지 변환 실패"
        RETURN

    # 1) 이미지 리사이즈
    RESIZE cv_image to (640, 480) as resized_color

    # 2) 그레이스케일 변환
    CONVERT resized_color to grayscale as gray_image

    # 3) ORB로 특징점 검출 및 디스크립터 생성
    DETECT keypoints and descriptors using ORB on gray_image

    # 4) 오버레이 페이드
    FADE overlay by multiplying with fade_factor

    # 5) 매칭 수행
    IF prev descriptors and descriptors exist:
        MATCH descriptors with prev descriptors using BFMatcher
        FILTER good_matches using ratio test
        SORT and select top N_MATCHES

        EXTRACT previous and current points from good_matches
        DRAW lines on overlay between matched points

    # 6) 이전 프레임 저장
    SET prev_keypoints to keypoints
    SET prev_descriptors to descriptors

    # 7) 오버레이 합성
    COMBINE resized_color and overlay using overlay_alpha as result

    # 8) 키포인트 시각화
    DRAW green circles on result for each keypoint

    # 9) 결과 표시
    SHOW result in OpenCV window

END image_callback
```

```
MAIN:
    INITIALIZE rclpy
    CREATE VisualTrackingNode
    TRY:
        SPIN node
    EXCEPT KeyboardInterrupt:
        LOG "Shutting down VisualTrackingNode."
    FINALLY:
        DESTROY OpenCV windows
        DESTROY node
        SHUTDOWN rclpy

END MAIN
```

Demo

감사합니다