0. 작성 시 주의사항

※아래의 작성 양식(제출분량, 폰트, 크기, 줄 간격 등)을 미준수 시 서류 평가의 감점요인됨

※ 제출 분량 : A4 용지 상세내용 포함 20 page 이내
※ 작성 양식 (폰트 : 맑은 고딕 / 폰트 크기 : 10pt / 자간 : 0% / 장평 : 100% / 줄 간격 : 130%)
※ 제출 포맷 : pdf

1. 팀 정보

팀명	혼파망	팀장	윤태현
팀원	황동하	팀원	

2. 개발완료보고서

1. 개요

1.1. 작품 개요

본 작품은 경기장의 전체 크기를 입력해 주면 출발 지점에서 교통 상황을 인식한 뒤 주행하는데 가장 적은 시간이 걸리는 루트를 자동으로 산출하고 돌발 상황이 일어난 도로를 우회하여 주행하는 기기입니다. 하드웨어는 인식 오류나 주행중의 오차를 최소화하기 위해 최대한 단순하고 작게 제작했으며, 소프트웨어는 경기장의 크기와 돌발 상황의 개수에 관계없이 교통정보만 있다면 주행이 가능한 구조로 제작하여 결선 당일 어떤 일이 발생하더라도 대처할 수 있도록 했습니다.

1.2. 개발 목표

가장 큰 목표는 결선 당일에 경기장에 어떤 변화가 생길지 알 수 없는 것을 감안하여 경기장의 크기, 돌발 상황의 개수와 관계없이 어떤 상황에서든 경기장이 직사각형이고 경기장에 존재하는 교차로의 개수만큼의 교통 정보가 주어진다면 주행할 수 있도록 하는 것입니다. 두 번째는 주행의 정확성 향상으로, 하드웨어를 가능한 한 단순화하고 소형화하여 하드웨어의 구조상 결함을 최대한 배제하고 주행 소프트웨어 개선으로 컬러 센서의 인식률을 높여 주행 중에 오차가 발생할 만한 상황을 줄이는 것입니다.

2. 개발 환경 설명 (최대한 자세하게 기술)

2.1. Software 구성

먼저 크게 나누면 경로 주행 부분과 경로를 구하는 부분으로 나누어져 있다고 할 수 있습니다. 그리고 이 둘을 포함하는 매인이 있지만 가장 주요한 부분은 이 둘이라고 할 수 있습니다. 경로를 구하는 부분은 Get_route_drive함수가 대표하고 있습니다. 이 함수는 주어진 경로의 시간을 구하는 부분(*1), route_first_suggestion을 구해주는 부분(*2), 경로의 주행 가능 여부를 판단하는 부분(*3), 이 셋을 이용하여 실제 주행할 경로를 구하는 부분으로 이루어져 있습니다.

(*1) Get_Time_InRoute

(*2) Get_route_first_suggestion

(*3) Check_NoEx

경로 주행 부분은 Drive_In_Route함수가 대표하고 있습니다. 이 함수는 주어진 경로를 주행하는 역할을 맡고 있습니다. 기본적으로 하위 함수들을 사용하여 교차로 단위로 주행(*1)합니다. 교차로 단위로 주행하는 것은 교차로로 진입하는 부분과 교차로 내에서 주행하는 부분(*2)으로 나누어집니다. 교차로로 진입하는 부분은 직진 코스에서 주행하는 부분(*3)과 우회전하는 부분(*4)으로 나누어져 있고, 교차로 내에서 주행하는 부분은 커브 코스에서 주행하는 부분(*5)과 커브 코스를 주행하기 전에 준비하는 부분(*6)으로 나누어집니다. 만약 갑작스러운 사고가 발생한다면 하위 함수에서 Drive_In_Route함수로 온 후, Drive_In_Route함수에서 다음 주행할 준비를 한 후 Get_route_drive 함수를 사용하여 다음에 주행할 경로를 구하고 자기 자신을 호출해 주행을 이

어나갑니다.

(*1) drive_in_two_crossway

(*2) drive_in_crossway

(*3) drive_until_black_straight

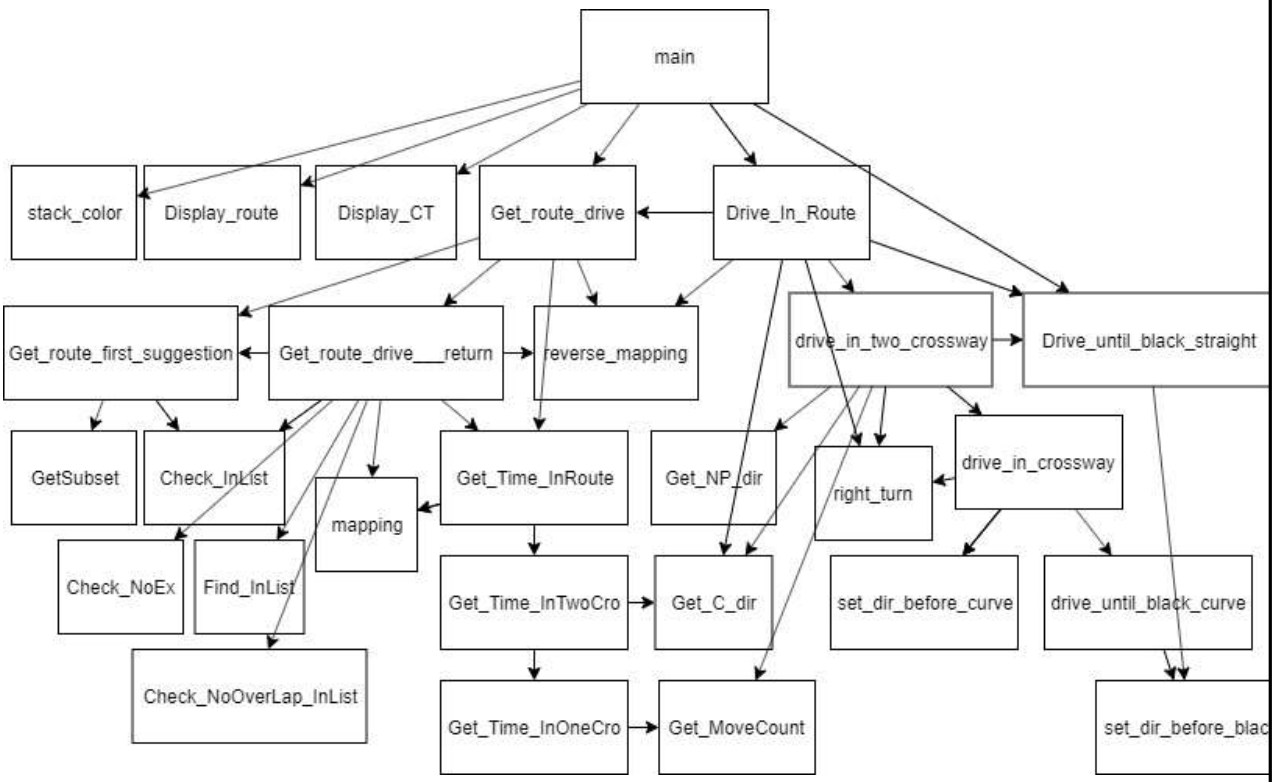
(*4) right_turn

(*5) drive_until_black_curve

(*6) set_dir_before_curve

2.2. Software 설계도(흐름도 및 클래스 다이어그램 등 (개발언어에 따라 선택))

밑의 그림의 사각형들은 프로그램의 함수들입니다. 몇몇 넣는 의미가 없다고 판단되는 함수들은 제외하였습니다. 화살표 A -> B는 A가 B를 호출한다는 것을 의미합니다.



2.3. Software 기능 (알고리즘 설명 포함)

먼저 이 프로그램의 기능은 챌린저 부문이니만큼 당연히 주어진 미션의 수행입니다. 그리고 이제 프로그램을 설명할 텐데 프로그램을 전부 다 설명할 수는 없으니 핵심적인 부분 몇 가지만 설명할 생각입니다.

정보 저장 방식

먼저 주행에 필요한 정보를 저장하는 변수들에 대해서 알아보겠습니다. cro(crossway) 는 교차로의 번호를 정보로 갖습니다. row, cul(culumn)은 교차로의 위치 정보를 좌표로 갖습니다. 따라서 cro와 나타내는 정보가 같습니다. 하지만 좌표로 저장하는 편이 사용하기에 더 편리한 부분이 있기 때문에 사용합니다. 1번 교차로의 좌표는 (0, 0)이고 3x3경기장 기준 9번 교차로의 좌표는 (2,2)입니다. 또한 dir(direction) 은 한 교차로 내에서 어느 위치의 검은색 원에 있는지를 저장하는 변수입니다. 1번 교차로의 왼쪽에서부터 시계 반대 방향으로 0, 1, 2, 3의 값을 갖습니다. 이 변수의 앞에 CP, NP 등이 붙기도 하는데, CP(current position)는 현재 위치의 정보임을 나타내고 NP(next position)는 다음으로 갈 위치의 정보임을 나타냅니다. C(center position)는 CP와의 구별을 위해 P를 생략했습니다. 이것은 dir앞에만 붙고, 현재 있는 교차로에서 그 교차로를 빠져나갈 때의 검은색 원의 위치를 나타냄을 의미합니다. 또한 이름에 route가 들어간 것들은 모두 경로를 나타냅니다. 이 경로는 교차로의 위치들로 표현되는데, 이 교차로의 위치들은 cro의 형태로 표현됩니다.

route_first_suggestion

route_first_suggestion은 색상 정보와 갑작스러운 사고를 고려하지 않고 처음으로 출발 지점과 맞닿아 있는 교차로를 지나며, 도착 지점과 맞닿아 있는 교차로를 지나고, 지나는 교차로의 개수가 최소인 모든 경로들을 의미합니다. 이 route_first_suggestion에 있는 경로만이 갑작스러운 사고를 고려하지 않을 때 최단 시간 경로가 될 수 있습니다. 이유는 다음과 같습니다. 먼저 최단 시간 경로가 될 수 있으려면 같은 위치를 2번 이상 지나지 않아야 합니다. 이것을 조금만 확장하면 같은 교차로에 2번 이상 진입하지 않는다는 뜻이 됩니다. 앞으로 교차로는 좌표로 나타내겠습니다. (0,0)에서는 (1,0) 또는 (0,1)이라는 2가지 선택지가 존재합니다. 만약 (1,0)으로 갔다면 (0,1)으로 가는 것은 최단 시간 경로가 아니게 되므로 (0,1)은 고려대상에서 제외됩니다. 이때 (0,1)이 고려대상에서 제외됨으로서 (0,2)로 가면 최단 시간 경로가 아니게 되므로 고려대상에서 제외되게 됩니다. 이어서 (0,3), (0,4) 등이 계속 제외됩니다. 따라서 (1,0)으로 감으로서 (0,a)는 전부 고려대상에서 제외되게 됩니다. 반대의 경우도 마찬가지입니다. 따라서 갑작스러운 사고를 고려하지 않았을 때는 최단 시간 경로가 될 수 있는 것은 route_first_suggestion에 있는 경로들뿐입니다. 단, 이는 전체 맵의 크기가 작을 때 성립합니다. 다만 이것이 성립하지 않을 정도로 맵의 크기가 늘어나지는 않을 것이라고 생각하기 때문에 이것을 바탕으로 프로그램을 구성하였습니다. 또한 route_first_suggestion은 갑작스러운 사고가 발생했을 때도 이것

을 바탕으로 주행할 경로를 구합니다. 이에 대해서는 Get_route_drive를 설명할 때 같이 설명하도록 하겠습니다.

주어진 경로를 주행할 때 걸리는 시간을 구하는 부분

이것은 Get_Time_InRoute를 설명하는 것과 같습니다. Get_Time_InRoute는 경로 정보와 현재 위치 정보가 필요합니다. 경로를 나누어 순서대로 한 교차로씩 시간을 구합니다. 이때 교차로의 위치 정보는 주어진 경로에 포함되어 있기 때문에 쉽게 구할 수 있습니다. 하지만 dir값은 현재 위치의 정보만 알기 때문에 별도의 과정을 거쳐서 알아내야 합니다. 그렇기 때문에 위치 계산은 dir값 계산이 대부분이 됩니다.

실제로 계산하는 과정은 간단합니다. 먼저 CP_row, CP_cul, NP_row, NP_cul을 알고 있으니 이것을 가지고 C_dir을 구합니다. 그리고 이제 CP_dir, C_dir을 알고 있으니 이것을 가지고 몇 번의 검은색 원 사이를 주행해야 하는지 구합니다. 그리고 이 값에 해당 교차로의 1/4만큼 돌 동안의 시간을 곱해 주면 한 교차로 내에서 걸리는 시간을 구할 수 있습니다. 그리고 이것을 반복하면 전체 경로의 주행에 걸리는 시간을 알 수 있습니다.

경로의 주행 가능 여부를 판단하는 부분

이 부분은 Check_NoEx함수가 담당하고 있습니다. 주행이 불가능하다면 그 경로를 배제하는 용도로 사용됩니다. 사실 주행 가능 여부 뿐만이 아니라 거의 명백하게 최단시간거리의 경로가 아닌 경로들도 배제합니다. 판단은 다음의 조건들에 대한 부합 여부를 판단하면 됩니다.

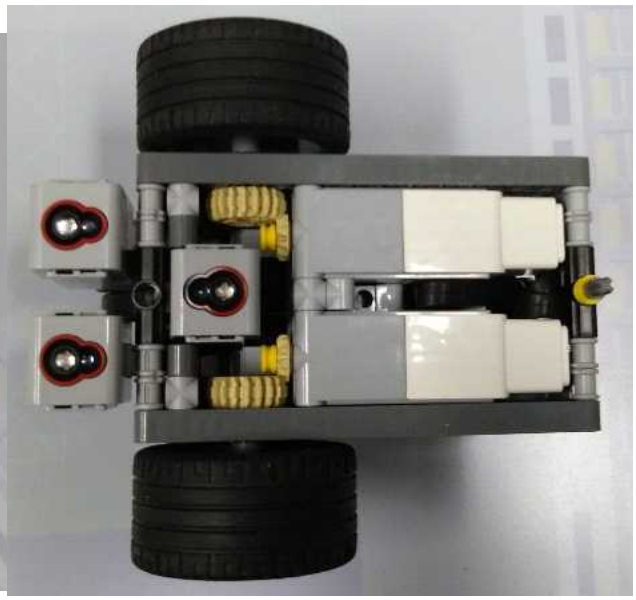
1. 한 교차로를 2번 이상 지나지 않는가?
2. 도착 지점이 경로에 포함되어 있다면 도착 지점이 경로의 가장 마지막에 위치해 있는가?
3. 경로 내의 교차로와 그 다음 교차로는 이웃하는가?
4. 경로 내의 교차로와 그 다음 교차로는 일치하지 않는가?
5. 경로 내의 교차로와 그 다음 교차로 사이에 사고가 발생하지 않았는가?

실제로 주행할 경로를 구하는 부분

이제 실제로 주행할 경로를 구하는 부분 즉 Get_route_drive를 알아보겠습니다. 먼저 이 함수는 갑작스러운 사고의 발생 여부를 판단합니다. 갑작스러운 사고가 발생하지 않았다면 route_first_suggestion에 최단 시간 경로가 포함되어 있기 때문에 이 안에서 최단 시간 경로를 구합니다. 갑작스러운 사고가 발생했다면 Get_route_drive__return함수를 호출하여 최단 시간 경로를 구합니다. 이 함수는 재귀함수입니다. MaxTurNum값을 1감소시켜 자기 자신을 호출하며, MaxTurNum값이 0이 되면 자기 자신의 호출을 중지하는 방식입니다. 이 함수는 MaxTurNum이 1인 상태로 호출되면 현재 위치에서 1번 상하좌우로 가는 경로를 모두 구하고, 이 경로의 마지막에서 route_first_suggestion인 경로와 합류하여 새로운 경로를 만들어냅니다. 그리고 이렇게 해서 만들어진 경로들의 주행 시간을 비교하여 이 중에서 최단시간경로를 뽑습니다. MaxTurNum은 maximum turn number즉 최대 회전 횟수를 의미합니다. 이 횟수는 상하좌우로 가는 경로를 만들어 내는 횟수를 뜻합니다. MaxTurNum이 2이상일 때는 그만큼 상하좌우로 간 경로들에서 또다시 상하좌우로 가는 경로를 만들어냅니다. 사고가 발생했을 때는 방향을 바꾸는 횟수도 늘어나야 최단 시간 경로를 찾지 못하는 경우가 없게 되기 때문에 MaxTurNum이 사고 발생 횟수에 비례해서 늘어나도록 하였습니다. 이론상 MaxTurNum을 충분히 큰 값으로 호출을 하게 되면 route_first_suggestion없이도 최단 시간 경로를 그것도 맵이 커

서 사고를 고려하지 않았을 때 route_first_suggestion에 최단 시간 경로가 없을 경우에도 구할 수 있습니다. 하지만 그렇게 하게 되면 시간이 너무나도 많이 걸리기 때문에 이러한 방식으로 최단 시간 경로를 구하는 것입니다.

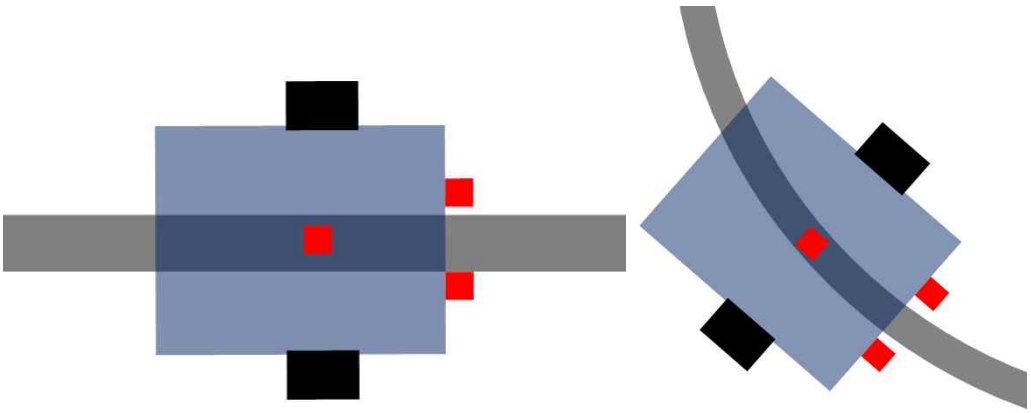
주행



주행을 설명하기에 앞서 컬러 센서를 정의하겠습니다. 로봇의 컬러 센서는 앞쪽에 2개, 로봇 중앙의 근처에 1개가 있습니다. 앞쪽의 컬러 센서 중 왼쪽에 있는 컬러 센서를 LCS, 오른쪽에 있는 컬러 센서를 RCS, 로봇 중앙의 근처에 있는 컬러 센서를 CCS라고 하겠습니다.

또한 프로그램에서 컬러 센서를 반사광 모드로 사용할 때는 컬러 센서 이름 뒤에 _RV을 붙인 이름의 함수를 호출하고, 컬러 모드로 사용할 때는 컬러 센서 이름 뒤에 _CV를 붙인 이름의 함수를 호출합니다. 컬러 센서 모드는 회색 선과 회색 선이 아닌 맵을 구분하지 못하기 때문에 주행 시에 회색 선을 인식하는 역할을 맡은 RCS와 LCS는 반사광 모드를 사용합니다. 하지만 CCS는 검은색을 정확히 인식하는 역할을 맡기 때문에 컬러 모드를 사용합니다. 예외적으로 색상 정보를 인식할 때는 모두 컬러 모드를 사용합니다.

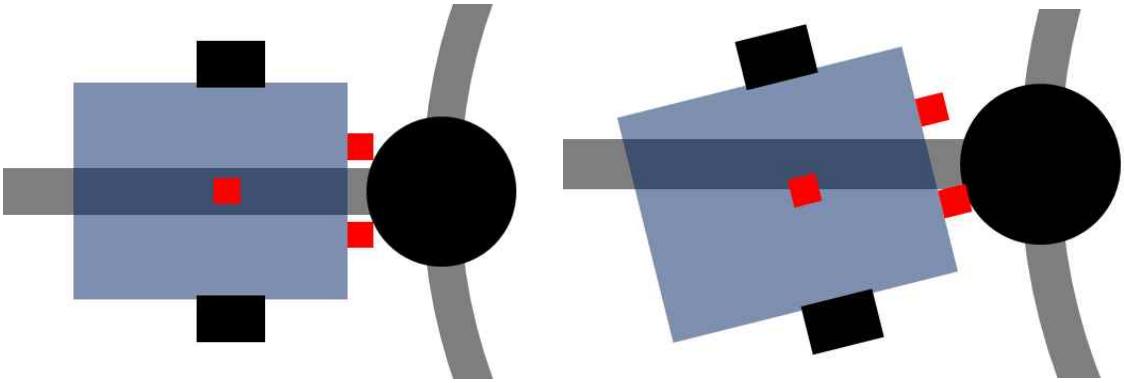
주행 알고리즘은 개발계획서에 서술한 것을 거의 그대로 가져왔습니다. 혹시라도 개발계획서를 보셨다면 보지 않으셔도 됩니다.



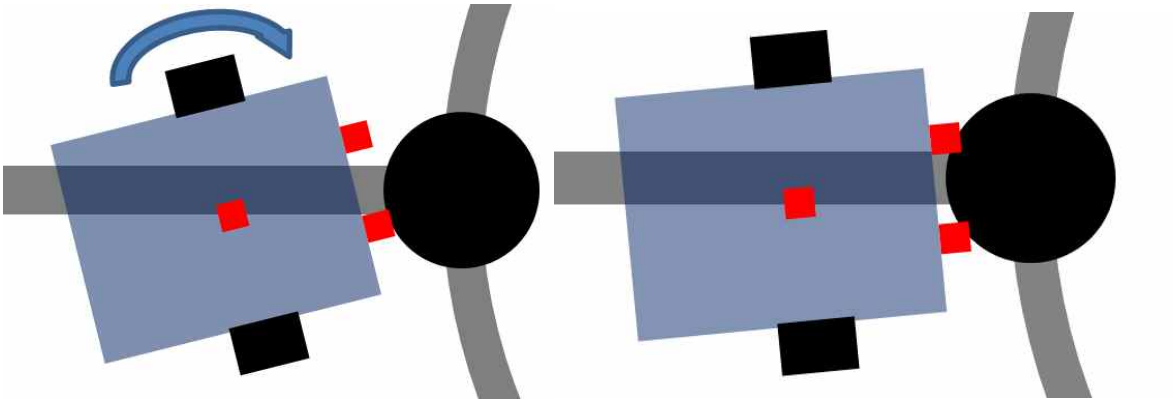
기본적인 직선코스와 곡선코스 주행입니다. 직선코스와 곡선코스 모두 라인에 맞춰서 주행하되, 라인에서 벗어날려고 하면 잡아주는 방식입니다.

자세히 설명드리자면, 직선코스는 기본적으로 직진합니다. 그때 컬러센서에 회색 선을 벗어나는 것이 감지되면 방향을 조금 조정해주는 식입니다. 곡선코스는 곡률을 알 수 있는 모두 같은 크기의 원을 도는 것이기 때문에 모든 곡선코스가 같습니다. 그러므로 곡선코스의 모양과 같은 모양으로 주행을 할 수 있습니다. 그러므로 기본적으로 곡선코스와 같은 모양으로 주행을 하고, 컬러센서에 회색 라인을 벗어나는 것이 감지되면 주행을 조금씩 조정해가면서 주행합니다. 이렇게 주행하면 빠른 속도로 주행하는 것이 가능합니다. 경로와 같게 주행하되, 잘못된 방향으로 가면 조금씩 조정해주는 것이기 때문입니다. 경로가 단순하기 때문에 이런 주

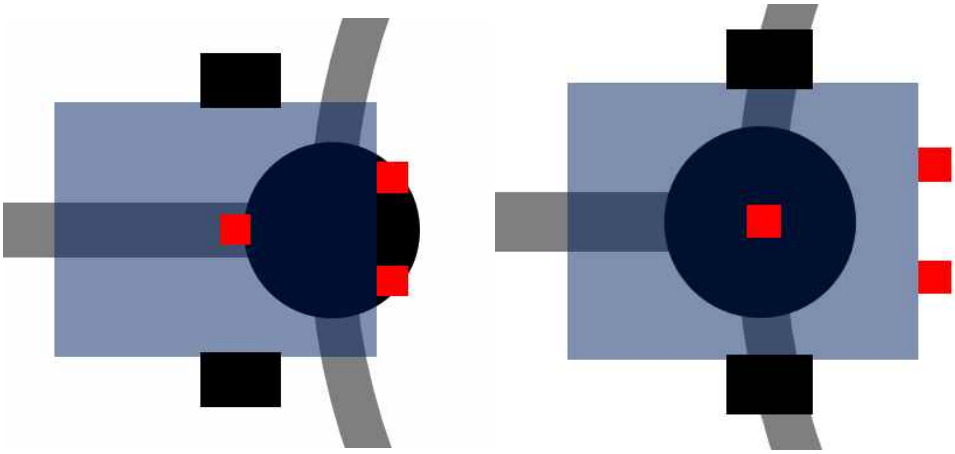
행을 할 수 있습니다. 다만 이렇게 주행하려면 직선코스&곡선코스에 진입하는 것이 정확해야 합니다. 직선코스 또는 곡선코스에 진입하는 경로에는 검은색 원이 위치해 있으므로 직선코스 또는 곡선코스에 진입하려면 검은색 원을 거쳐야 합니다. 따라서 직선코스&곡선콧에 진입하는 것이 정확해지려면 검은색 원을 정확히 통과해야 합니다.



직선코스에서 곡선코스로 진입하는 것을 먼저 설명해보겠습니다. 직선코스에서 주행하다가 검은색 원을 만날 때 직선코스에서 곡선코스로 진입하는 것이 시작됩니다. 이때 진입하는 것이 이렇게 틀어질 수도 있습니다.



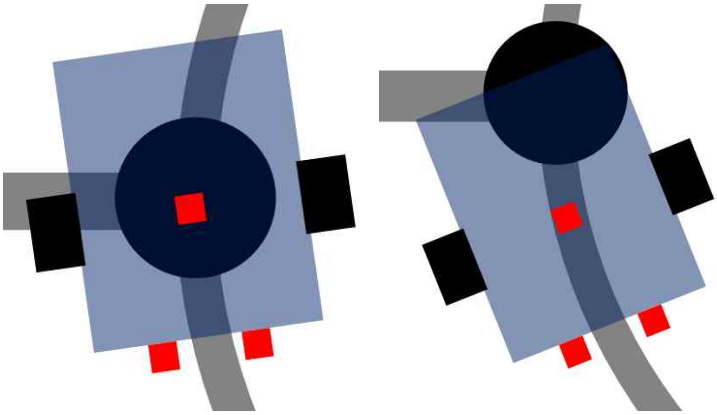
만약 검은색 원에 진입하는 것이 틀어진다면 교정을 해줘야 합니다. 만약 RCS만 검은색을 감지하면 왼쪽 바퀴만을 LCS가 검은색을 인식할 때까지 움직여서 틀어진 것을 교정해줍니다. 반대로 LCS만 검은색을 감지하면 오른쪽 바퀴만을 RCS가 검은색을 인식할 때까지 움직여서 틀어진 것을 교정해줍니다. 이렇게 하였음에도 충분히 교정되지 않았다면 약간 뒤로 물러나서 다시 검은색 원에 진입해야 합니다.



그 후에는 CCS가 검은색을 인식할 때까지 직진합니다. CCS가 검은색을 인식하면 CCS가 검은색 원의 중앙에 위치할 때까지 *정해진 거리만큼 직진합니다. CCS가 검은색 원의 중앙에 위치하게 되면 회전할 때의 회전축이 CCS가 되기 때문에 회전하여 곡선코스로 넘어가기 수월해집니다.

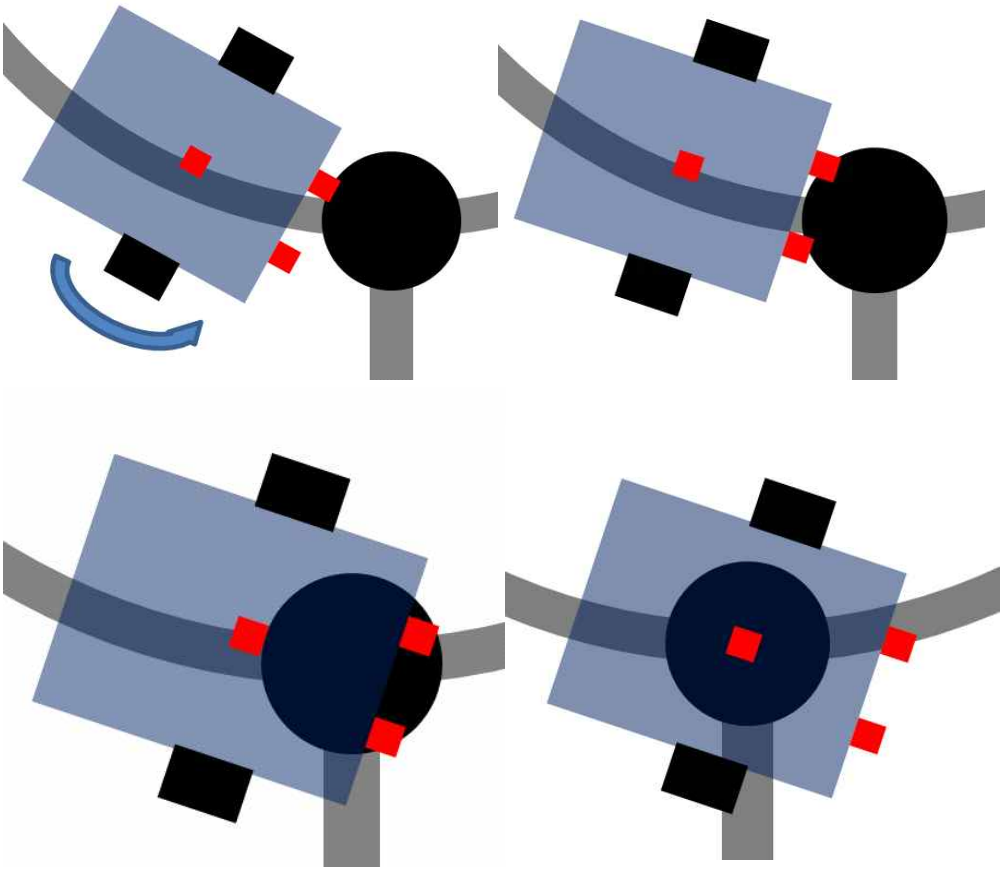
*여기에서 모터 센서가 사용되어 정해진 거리만큼 직진하게 합니다. 모터 센서를 사용하지 않고 정해진 시간만큼 직진하게 하면 배터리 잔량에 따라서 직진하는 거리가 달라집니다.

그렇기 때문에 모터 센서를 사용하여 정해진 거리만큼 직진하게 합니다.



우회전을 합니다. LCS와 RCS를 이용하여 LCS와 RCS 사이에 회색 선(곡선 코스)이 들어오도록 우회전을 합니다. 그 후 곡선코스 주행을 하면 됩니다.

다음으로는 곡선코스에서 직선코스 또는 곡선코스로 진입하는 것입니다.



(직선코스에서 곡선코스로 진입하는 과정과 매우 흡사합니다. 거의 똑같다고 봐도 됩니다. 그림으로만 봐도 쉽게 이해할 수 있을 것 같으므로 굳이 설명할 필요는 없다고 생각합니다.)

이렇게 CCS가 검은색 원의 중심에 위치하게 됩니다. 여기서 곡선코스로 가려면 좌회전하여 LCS와 RCS 사이에 회색 선(곡선코스)을 위치하게 한 후 곡선코스 주행을 하면 됩니다. 직선코스로 가려면 우회전하여 LCS와 RCS 사이에 회색 선(직선코스)을 위치하게 한 후 직선코스 주행을 하면 됩니다.

2.4. 프로그램 사용법 (Interface)

"Download and run"

출발할 때 직진해서 색상 정보를 나타내는 테이프에 앞의 두 컬러 센서가 모두 올려지도록

한 후 프로그램을 실행하면 됩니다.

2.5. 개발환경 (언어, Tool, 사용시스템 등)

기술지원세미나에서 지원해준(알려준) 파이썬 프로그램인 'vscode'입니다.
ev3dev-browser, LEGO MINDSTORMS EV3 MicroPython, Python for VSCode

3. 개발 프로그램 설명 (최대한 자세하게 기술)

3.1. 파일 구성

Testeueueu
.vscode
extensions.json
launch.json
settings.json
.gitignore.gitignore
main.py

3.2. 함수별 기능

display_color_sensor()
main 에는 들어가지 않는 함수로, 컬러 센서의 값을 확인하는 용도로 사용합니다.
MotorStop(time)
주행 사이사이에 모터를 잠시 멈추는 용도로 사용합니다.
set_dir_before_black()
교차로에 4개 있는 검은색 원에 진입하는 용도로 사용합니다.
set_dir_before_curve()
곡선 경로 주행 전에 실행하여 곡선 경로 주행 준비용으로 사용합니다.
drive_until_black_straight()
직선 경로 주행에 사용하는 함수입니다.
drive_until_black_curve()
곡선 경로 주행에 사용하는 함수입니다.
right_turn()
우회전을 하는 함수입니다.
stack_color()
색상 정보를 인식하는 용도로 사용하는 함수입니다.
Display_CT()
CT는 crossway time의 약어이며, 색상 정보를 저장하는 리스트입니다.
이 함수는 CT를 브릭에 출력하는 함수입니다.
Display_route(route)
경로를 브릭에 출력하는 함수입니다.
drive_in_crossway(count)
한 교차로 내에서 이동을 담당하는 함수입니다.
drive_in_two_crossway(CP_row, CP_cul, CP_dir, NP_row, NP_cul)
교차로로의 진입과 drive_in_crossway함수를 사용하여 진입한 교차로 내에서의 이동을 하는 함수입니다.
Drive_In_Route(route,CP_dir = 0)
주어진 경로를 주행하는 함수입니다. 대부분의 주행 함수들을 포함합니다.

그리고 갑작스러운 사고가 발생했을 때 주행할 경로를 알려주는 함수를 호출하고
그로부터 받은 경로를 가지고 자기 자신을 호출하여 주행을 계속합니다.

mapping(cro)

교차로의 위치가 교차로 번호로 저장되어 있는 것을 좌표로 바꾸어 주는 함수입니다.

reverse_mapping(row, cul)

교차로의 위치가 좌표로 저장되어 있는 것을 교차로 번호로 바꾸어 주는 함수입니다.

Check_InList(input_list, value)

특정 리스트 내의 특정 값의 존재 유무를 알려주는 함수입니다.

Check_NoOverlap_InList(input_list)

리스트의 중복되는 값의 존재 유무를 알려주는 함수입니다.

Find_InList(input_list, value)

특정 리스트 내의 특정 값의 위치를 알려주는 함수입니다.
보통 Check_InList함수로 존재하는 것을 알아낸 후에 사용합니다.

GetSubset(Lst)

리스트의 부분집합에 해당하는 리스트를 모두 구해주는 함수입니다.

Get_route_first_suggestion()

route_first_suggestion를 구해주는 함수입니다.

Get_MoveCount(CP_dir, C_dir)

한 교차로 내에서 몇 번 검은색 원들의 사이를 움직여야 하는지를 구하는 함수입니다.

Get_C_dir(CP_row, CP_cul, NP_row, NP_cul)

C_dir을 구하는 함수입니다.

Get_NP_dir(C_dir)

NP_dir을 구하는 함수입니다.

Get_Time_InOneCro(CT_value, CP_dir, C_dir)

한 교차로 내에서 주행하면 얼마만큼의 시간이 걸릴지를 알려주는 알려주는 함수입니다.

Get_Time_InTwoCro(CP_row, CP_cul, CP_dir, NP_row, NP_cul)

사실상 Get_Time_InOneCro함수를 호출하기 위한 중간 과정으로 사용하는 함수입니다.

Get_Time_InRoute(route, CP_dir = 0)

주어진 경로의 시간을 구해주는 함수입니다.

Check_NoEx(route)

주어진 경로가 주행 가능한지, 2번 이상 지나는 교차로가 없는지, 도착 지점이 경로에
포함되어 있다면 경로의 마지막에 있는지의 여부를 알려주는 함수입니다.

Get_route_drive(CP_row, CP_cul, CP_dir = 0)

실제로 주행할 경로를 구하는 함수입니다.

Get_route_drive___return(CP_dir, route_drive, route_drive_time, C_route_drive, MaxTurNum)

갑작스러운 사고가 발생했을 때Get_route_drive에서 호출되어 실질적으로 갑작스러운 사고를 고려한 경로를 구하는 함수입니다. Get_route_drive에 포함되어 있고 재귀함수를 사용한다는 뜻으로 이름을 이렇게 지었습니다.

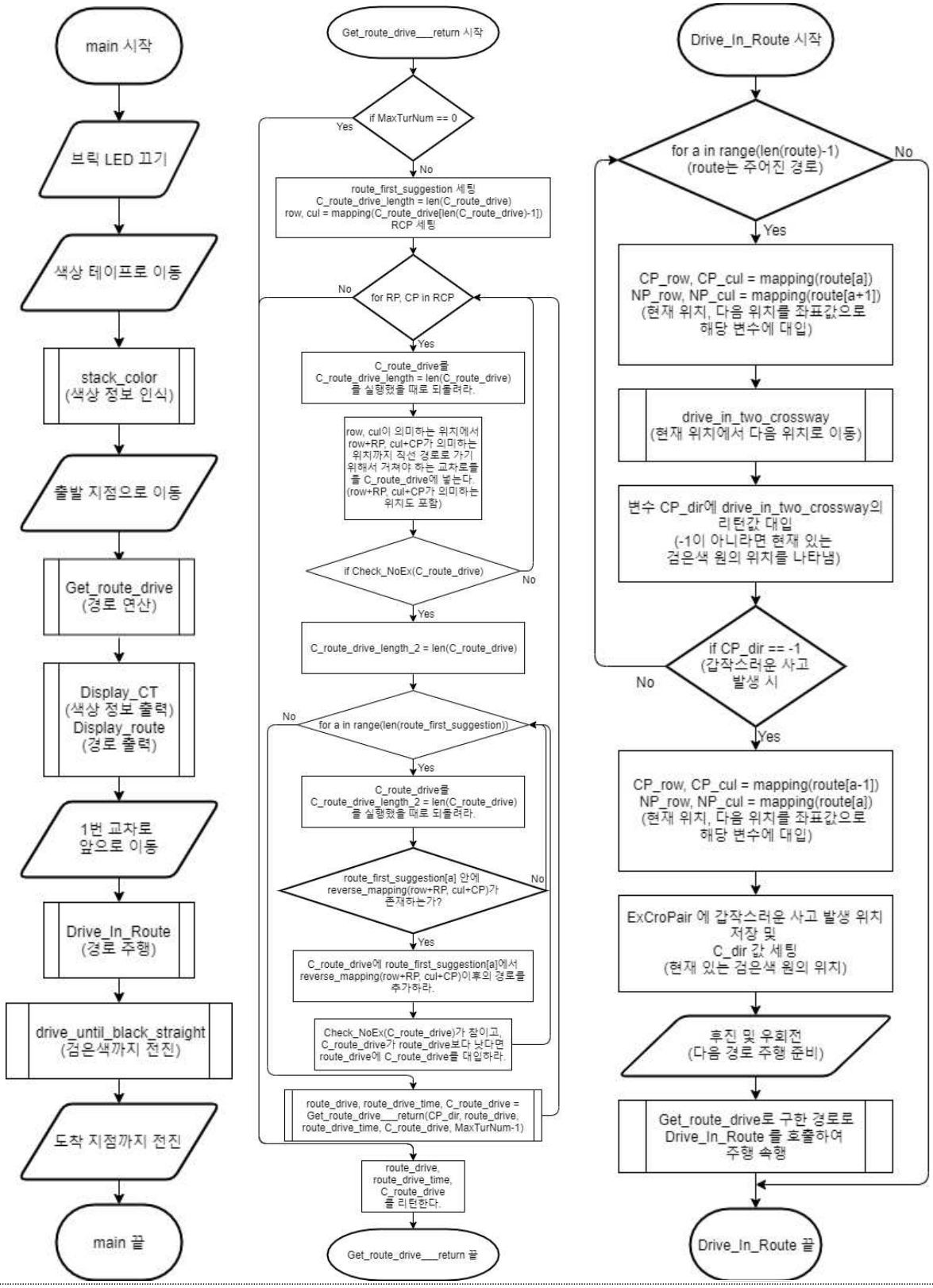
main()

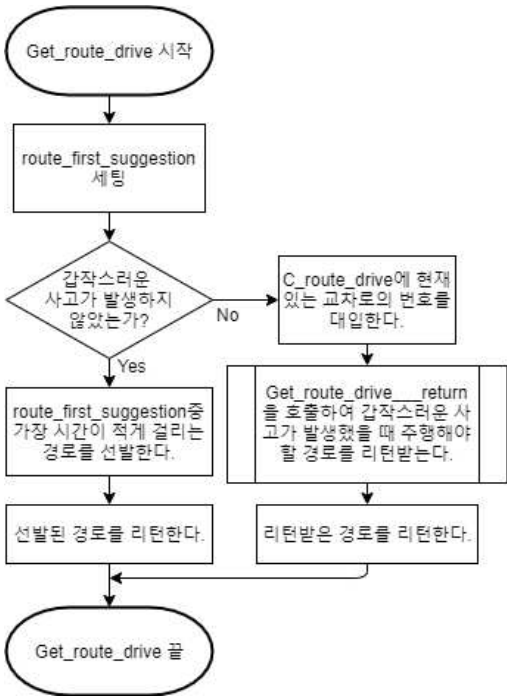
모든 함수가 포함되어 있으며 말 그대로 매인입니다.
실행할 때는 그저 이 함수만을 호출하면 됩니다.

3.3. 주요 함수의 흐름도

먼저 모든 함수를 아우르는 main함수가 있고, 이를 제외하면 가장 주요한 함수는 Get_route_drive와 Drive_In_Route입니다. Get_route_drive는 실제로 주행할 경로를 구하는 함수로서 경로를 얻기 위한 모든 함수를 포함합니다. Drive_In_Route는 주어진 경로를 주행하는 함수입

니다. 따라서 주행을 위한 함수를 포함하고 돌발 생황이 발생했을 때는 Get_route_drive함수까지 포함하기 때문에 정말로 중요한 함수라고 할 수 있습니다.





3.4. 기술적 차별성

다른 팀들이 어떻게 만들었는지 모르기 때문에 확실하게 차별성이 있다고 말하기 힘들지만, 경기장의 모양이 직사각형이고 출발과 도착이 직사각형의 양 끝이며, 갑작스러운 사고로 모든 경로가 막혀있지 않다면 map_row와 map_cul*만 바꿔주면 정상적으로 미션을 수행할 수 있다는 점입니다. *map_row와 map_cul은 각각 맵 자체의 가로, 세로의 길이를 나타내는 전역변수입니다. route_first_suggestion은 색상 정보와 갑작스러운 사고를 고려하지 않고 처음으로 출발 지점과 맞닿아 있는 교차로를 지나며, 도착 지점과 맞닿아 있는 교차로를 지나고 지나는 교차로의 개수가 최소인 모든 경로들을 의미합니다. 이 route_first_suggestion에 있는 경로만이 갑작스러운 사고를 고려하지 않을 때 최단시간경로가 될 수 있고, 갑작스러운 사고가 발생했을 때도 이것을 바탕으로 주행할 경로를 구합니다. 필자가 생각하기에 route_first_suggestion이 경로 계산의 토대가 된다는 것이 다른 경로 계산 프로그램과의 가장 큰 차이입니다.

4. 개발 중 장애요인과 해결방안

개발 중 가장 컸던 장애물은 최초 주행 루트와 돌발상황 발생 시 새로운 루트를 구하는 Get_route_drive와 Get_route_drive__Return(재귀함수) 함수의 개발로, 최종 소프트웨어의 다른 어떤 함수보다도 알고리즘의 설계 난이도가 극단적으로 높고 다종다양한 버그가 발생해 해당 함수의 개발로만 개발 시간의 반 정도를 소모하게 되었습니다. 장애물을 극복한 방법은 단순하게 버그가 발생하면 해결하고, 결과값에 오류가 발생하면 문제점을 찾아 알고리즘을 수정하고 새로운 버그를 만들어내며 수도 없이 많은 수정을 거듭하여 결국 함수를 완성할 수 있었습니다. 두 번째로는 Get_route_first_suggestion 함수로, 교통상황에 관계없이 (3*3기준) 1번 교차로와 9번 교차로를 지나고 지나는 교차로의 개수가 5인 모든 루트를 포함하는 리스트를 생성하는 함수입니다. 이 함수를 개발할 때 "어떤 리스트에서 길이가 n인 모든 부분집합을 구하는 알고리즘"이 필요했으며, 이를 구글링을 통해 해결하려 하였으나 예상보다 자료가 부족하여 자료 조사에 애를 먹었습니다. 마지막으로 개발에 난항을 겪었던 점은 광량에 따라 변하는 컬러 센서의 리플렉션 값과 모터의 부정확성에 의한 주행 인식 오류와 탈선이었습니다. 경기장 뒤의 배경에 의해 컬러 센서가 방해를 받는 일도 비일비재했으며 모터의 정지가 제대로 이루어지지 않는 등의 문제가 있었습니다. 이러한 문제들은 라인트레이싱 알고리즘의 수정과 하드웨어 구조 변경으로 대부분 해결되었지만, 현재 디바이스가 라인을 벗어나는 확률을 0에 수렴하게 하는 것은 불가능했습니다.

5. 개발결과물의 차별성

기본적으로 경기장이 직사각형 형태이고 출발 지점과 도착 지점이 해당 직사각형의 대각선 끝에 있으며 모든 교차로에 대한 교통 정보가 주어진다면 도착지점에 도달하는 모든 루트가 막히지 않

은 상황에서 경기장의 크기, 가로 세로 비율, 돌발 상황의 개수와 관계없이 주행할 수 있습니다. 이는 주최 측에서 결선 당일 경기장에 변동이 있을 수 있다고 하였으므로 매우 큰 이점이 됩니다. 또한, 본선 때를 기준으로 했을 때 디바이스의 연산 속도가 압도적으로 빨랐습니다. 현위치로부터 뻗어나가는 루트를 구하고 해당 루트의 주행 시간을 구한 뒤 다른 루트로 넘어가는 방식이 아니라 경기장의 크기를 고려해 주행할 수 있는 모든 루트를 상정한 뒤 그 중 주행에 걸리는 시간이 가장 짧은 루트만을 추려내는 방식이기 때문에 가능한 것으로 보입니다. 마지막으로 디바이스의 크기가 매우 작습니다. 대회 규정으로 제한된 길이*폭*높이 25*25*30(cm)을 준수하는 것을 넘어, 미디엄 모터와 베벨 기어 등을 활용하고 센서를 동체 안으로 집어넣어 디바이스의 크기를 최대한 줄인 결과 14.5*13.5*10.5(cm)의 극단적으로 작은 사이즈에 주행 정확성을 유지하며 필수 기능을 모두 탑재할 수 있었습니다.

6. 단계별 개발계획 및 실제 참여인원 및 업무 분장

팀 인원이 총 2 명으로 적은 편이었지만, 오히려 인원이 적은 만큼 업무 분배를 간단하게 할 수 있었습니다. 개발 과정에서 손이 비는 인원이나 그로 인한 팀원간의 분쟁 등이 발생하지 않았던 것 또한 인원이 적은 것의 장점이었습니다. 따라서 효율적인 업무 분담을 위해 개발을 시작하기 전 대략적인 개발 순서를 계획할 때 하드웨어 개발과 소프트웨어 개발을 동시에 진행하도록 계획하였습니다. 팀장은 최단시간경로 계산 소프트웨어를 주로 담당하여 알고리즘을 설계, 프로그램화 하였고 팀원은 하드웨어를 설계 및 제작하고 라인트레이싱 소프트웨어를 제작한 뒤 팀장의 작업에 합류하여 버그 수정 등 팀장의 업무를 보조하였습니다.

S/W개발:

소프트웨어 개발의 대략적인 절차는 이러합니다. (위의 함수별 기능 설명에 각 함수의 자세한 역할이 서술되어 있으므로 자세한 설명은 생략합니다.)

1. 정보 저장 방식 확립
2. mapping, Get_C_dir 등의 기본적인 함수 제작
3. Get_Time_InRoute및 그에 포함, 관련된 함수들 제작
4. route_first_suggestion및 그에 포함, 관련된 함수들 제작
5. Get_route_drive및 그에 포함, 관련된 함수들 제작

본선 3*3 크기의 경기장에서 사용했던 프로그래밍 단계에서 미리 몇몇 루트를 지정해 둔 뒤 해당 루트의 주행에 걸리는 시간만을 비교하는 방식을 개량하여, route_first_suggestion 함수를 이용해 미리 입력받은 경기장의 가로와 세로 크기를 기반으로 주행 가능한 루트들을 생성하고 저장해 두는 방식으로 변경하였습니다. 다른 변경사항으로는 돌발 상황에 제대로 대처하지 못했던 본선 때와 달리, 알고리즘을 강화하여 돌발 상황의 개수에 관계없이 주행할 수 있도록 한 것이 있습니다.

H/W개발:

하드웨어를 개발할 때 한 가지 구조만을 계속 연구하는 것이 아니라 다양한 구조의 로봇을 제작하고 그 중 가장 정확성이 높은 구조를 채택하여 보완하는 방식으로 개발하였습니다. 개발 초기 설계한 로봇들은 현재 사용중인 미디엄 모터 외에 라지 모터를 사용한 형태가 있었으며, 같은 모터를 사용한 로봇들의 대략적인 형태는 비슷했지만 컬러 센서의 위치를 계속 변경하여 인식률을 높이는 것에 집중하였습니다. 해당 과정에서 최종적으로 채택된 구조는 미디엄 모터와 베벨 기어로 동체 크기를 최소화하고, 베벨 기어 사용으로 발생한 공간에 돌발상황 인식용 컬러 센서를 집어넣는 구조였습니다. 그렇게 하드웨어 제작을 완료한 뒤에는 라인트레이싱과 그 외의 주행용 소프트웨어 제작에 착수하였습니다. 하지만 해당 과정에서 컬러 센서 자체의 좋지 않은 인식률 때문에 개발에 난항을 겪었습니다. 색상 인식에는 큰 문제가 없었으나 리플렉션 값이 주변 광량의 변화에 무시할 수 없는 영향을 받아 정확한 센서값의 범위를 정하는 것에 예상보다 많은 시간을 빼앗겼습니다. 여러 번의 시도 끝에 센서값을 특정한 다음에는 본격적인 라인트레이싱 소프트웨어 제작에 착수하였습니다. 라인트레이싱 소프트웨어를 제작할 때 가장 중요시한 것은 당연하게도 주행의 정확성이었습니다. 로봇이 주행할 때 정확성을 가장 크게 저하시키는 것은 모터가 회전을 시작하거나 정지할 때 발생하는 모터와 바퀴의 정지 각도 차이입니다. 원래는 이 문제를 해결하려면 로봇이 동작을 시작하기 전에 앞이나 뒤로 움직인 다음 급정지해 좌우 바퀴의 각도를 같게 정렬해야 하지만, 하드웨어 구조에 기어를 사용한 덕분에 유격이 많이 줄어들어 소프트웨어를 제작할 때 무시해도 될 정도의 오차만이 남았습니다. 따라서 소프트웨어를 제작할 때 하드웨어 자체의 오차를 고려할 필요가 줄어 컬러 센서 때문에

낭비한 시간을 어느정도 만회할 수 있었습니다. 또한 개발 중 나름 공을 들였던 부분으로는 로봇이 교차로 내에서 이동하며 각 포인트에 진입할 때 방향을 정렬하는 부분이었습니다. 로봇 전면의 두 라인트레이싱용 컬러 센서 중 하나라도 검은색을 인식할 때까지 주행한 후 검은색 원에 완전히 진입하기 전 방향을 정렬하여 진입하는 방식으로 제작하게 되었는데, 이때 컬러 센서가 검은색을 인식한 뒤 로봇이 회전을 멈추기까지의 반응이 느려 불가피하게 해당 부분의 모터 속도를 낮출 수 밖에 없었습니다. 그 외에, 라인트레이싱 소프트웨어를 제작할 때 가장 큰 틀로 잡았던 것은 각도 기반 회전을 최소한도로 줄이고 센서의 인식을 기반으로 한 회전을 사용하는 것이었습니다. 이 방식은 상술했듯 기기 자체의 반응 속도 문제로 인해 주행 속도를 늦춰야 한다는 단점이 있지만, 각도 기반 회전에 비해 주행 중 발생하는 오차에 대한 대응력이 월등히 높다는 장점이 있어 부드럽고 정확한 주행에 도움이 되었습니다.