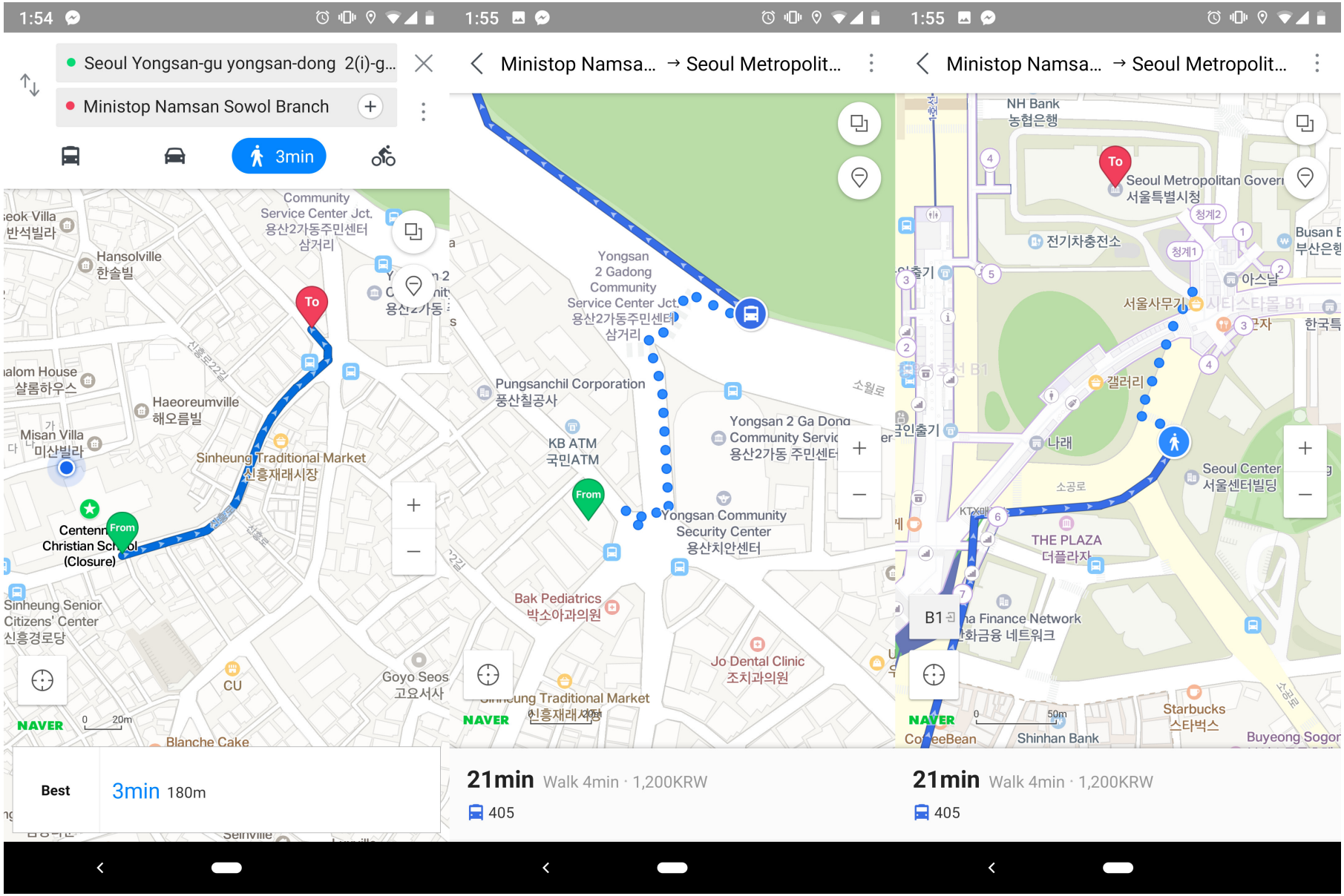
CS110 LBA

1. The two landmarks and intended route

1. Ministop near our residence hall

2. Seoul City Hall

Intended path:



2. The navigation algorithm

In [ ]:
```
robotAmal = Robot('amal', 'home', 0)

destinations = [
    {
        name: "Ministop"
        checkpoints: None
        distance: 0.3 # miles
    },

    {
        name: "City Hall"
        checkpoints: ["bus"]
        distance: 2.5 # miles
    }
]

robotAmal.exit_res_hall()

# first destination
# greedy by finding max evelation angle at each step
while robotAmal.cur_loc != destinations[0].name:
    cur_elev = robotAmal.measure_elevation()
    while robotAmal.measure_elevation() < cur_elev or robotAmal.current_image() != 'sidewalk':
        turn_left(10) # turn a bit
    robotAmal.walk(10)
    if robotAmal.current_image() == 'convenience store':
        robotAmal.refill_money(1200)
        robotAmal.update_loc(destinations[0].name)

    if robotAmal.travel_time() > 10:
        robotAmal.return_home()

robotAmal.congrats(0)


# second destination
while robotAmal.cur_loc != destinations[1].name:
    while robotAmal.current_image != 'stairs':
        cur_elev = robotAmal.measure_elevation()
        while robotAmal.measure_elevation() < cur_elev or robotAmal.current_image() != 'sidewalk':
            turn_right(10) # turn a bit
```

```
        robotAmal.walk(10)

    robotAmal.walk_up_the_stairs()
    robotAmal.walk_until('bus stop')
    robotAmal.turn_left(90)
    robotAmal.walk_until('sidewalk')

    wait_until('bus')
    while robotAmal.bus not in ['402', '405']:
        wait_until('bus')

    robotAmal.walk_up_the_stairs()
    print(robotAmal.capture_image()) # take a picture at checkpoint

    # greedy seat finding by finding best seat at current point
    # we assume that the robot can only observe one seat at a time and needs to get
    # to the end of the bus to get closer to the exit door
    while robotAmal.current_image() == "empty seat":
        robotAmal.walk_until('empty seat')
        # see the next empty seat

    if robotAmal.current_image("big glass building"):
        robotAmal.tap_card('T-money')
        robotAmal.walk_until('sidewalk')

    if robotAmal.travel_time() > 60:
        robotAmal.return_home()

robotAmal.congrats(1)
robotAmal.return_home()
```

3. The execution algorithm

In [ ]:

```
'''
Pseudo Libraries
'''
import fakeML # neural networks library
import life # built-in human operations
import eyes # visual signal library
import numpy # normal numpy
import time

class Robot:
''' Private methods (low-level unimportant details, public methods are more interesting to look at) '''
    self.__classes = ["bus stop", "bus", "empty seat" "crossroad", "sidewalk", "road", "green light",\
                      "door", "wall", "stairs", "big glass building"]
    self.__congrats_messages = \
    ["Great Job, {0}. Before we move on to the second, please refill your T-money transportation card"\
     .format(self.name),
     "Congratulations, you have fulfilled the mission. Enjoy your time there!"]

    def __import_classifier_weights(self):
        # self.weigths ⊂ R³
        self.weights = open("conv-neur-net-weight.xml")

    def __get_image():
        img = eyes.capture_image()
        return fakeML.transform_to_matrix(img) # returns np array ⊂ R*R*3 storing colors per pixel

    def __classify_image(self, bus=None):
        # forward pass and classification
        X = self.__get_image() # input matrix X ⊂ R²
        cur_activation_layer = fakeML.ReLU(X.dot(weights[0])) # dot product and non-linear transformation
        del weights[0]

        while weights:
            cur_activation_layer = cur_activation_layer.dot(weights[0]) # passing down the network
            del weights[0]

            if len(weights) != 0:
                cur_activation_layer = fakeML.ReLU(cur_activation_layer) # non-linearity
            else:
                probs = fakeML.SoftMax(cur_activation_layer) # assign probability scores
```

```
        return self.classes[np.argmax(probs)] # return class with highest probability


''' Public methods (interface) '''
    # robot initialization
    def __init__(self, name = 'untitled', initial_location = 'home', initial_orientation = 0):
        self.name = name

        # private instances accessible ONLY within class
        self.__cur_loc = initial_location
        self.__cur_orient = initial_orientation
        self.__import_classifier_weights()
        self.__timer = time.time()
        self.__life = life.Life() # initialize life-essential operations

    # get/set methods
    def cur_loc(self):
        return self.__cur_loc

    def update_loc(self, new_loc):
        self.__cur_loc = new_loc

    def current_image(self):
        return __classify_image(self)

    def travel_time(self):
        time_elapsed = time.time() - self.__timer
        return time_elapsed

    def cur_bus(self):
        self.__classify_image('bus')

    def measure_len(self, until):
        start = self.travel_time()
        while self.__capture_image() != until:
            walk(1)
        end = self.travel_time()

        # assuming that the robot walks at 5 km/h
        return (end - start) * 5

    def measure_elevation(self):
```

```
        img = self.__capture_(image)
        # because we look at the road from the top, then the top row of pixels is the end point
        actual_len = self.measure_len(image[0][:])
        # silly angle finding
        return arcsin(img.height / actual_length)

    # navigation execution
    def exit_res_hall():
        while self.__classify_image() != road:
            while self.__classify_image() == "wall":
                # in a room there are 3 walls and 1 door, and we might not know which wall we're looking at
                self.turn_right(90)

            if self.__classify_image() == "door":
                self.__life.open_door()

            self.walk_until(wall)

    def turn_left(self, deg):
        new_orient = self.init_orient - deg
        self.__cur_orient = new_orient

    def turn_right(self, deg):
        new_orient = self.init_orient + deg
        self.__cur_orient = new_orient

    def wait_until(self):
        while self.__classify_image() != "green light":
            self.__life.patiently_wait()

    def walk_until(self, obj):
        while self.__classify_image() != obj:
            cur_obj = self.__classify_image()
            if cur_obj != "crossroad":
                self.__life.follow_sidewalk('1m')

            elif cur_obj == "crossroad":
                self.wait_until("green light")
                while self.__classify_image() != "sidewalk":
                    self.walk(10)

    def refill_money(value):
```

```
        self.__life.deposit_money('T-money', value)

    def tap_card(card_type):
        self.__life.tap_card(card_type)

    def walk(self, dist=0):
        self.__life.walk(dist)

    def walk_up_the_stairs():
        while self.__classify_image() == "stairs":
            self.__life.walk(10)

    def return_home(self):
        app = open("Naver Maps")
        app.search("Centennial Christian School (Closure)")
        app.search_results[0].click()
        app.click("To")
        app.click("bus")
        route = eyes.capture_image(app.screen)

        # use life-learned ability to use maps to navigate home
        life.navigate(route, route, self.__cur_loc, self.__cur_orient)

    def congrats(self, loc_idx):
        print(self.__congrats_messages[loc_idx])
```

4. Amalanand's pseudocode

In [ ]:

```
'''
Navigation algorithm
'''
finalDestination = "Noryangjin"

follow_signs(subway station gates, enter )

while not at destination station,
    which_line_to_take(finalDestination)
  alighttrain(finalDestination)

follow_signs(Exit 3, exit)


'''
Execution algorithm
'''
def follow_signs(location,direction):
  if direction is exit:
    tap out of fare gates

  while sign to location not yet spotted
    walk() till a sign with direction to location is spotted
  follow signs to location

  if direction is enter:
    tap into fare gates
  else if direction is train:
    enter train



def walk():
  Take a step
  if facing wall, turn right 90 degrees
  if exited building into outdoors, turn around 180 degrees, re-enter door/exit
    and walk() 5 times


def which_line_to_take(destination):
  take out notebook[]
```

```
  find nearest subway map which you can reach
  current station = station name, which is obtained from station signs or the map

  put bottom of left index finger on current station
  put tip of right index finger on destination

  while maintaining posititon of bottom of left index finger, rotate top of left
      index finger till it points at right index finger

  for each subway line at current station:
    find acute angle between left index finger pointing at destination and the
    line on the map

    record down line number, direction and angle in notebook

  go through entries in notebook to select line number and direction with lowest angle

  follow signs(lineNumber, train)

  cross out all new entries in notebook


def alighttrain(destination):
  if train speaker says "Interchange" or "You can transfer to line" or destination:
    alight at next stop
```

5. The path I took

1. My starting point was Seoul Station at the light blue line. In step two, according to the algorithm (the `which_line_to_take()` function), I measure the angle from the straight line to the end station to other lines with the same starting point and find out that the dark blue line has the smallest angle of all. Therefore, I `follow_signs()` to transfer to Seoul Station on dark blue line.
2. After I arrive at Yongsan station, I alight the train because I heard that there will be an interchange. I measured the angles again and find out that the dark blue line still has the smallest angle, thus, its locally most optimal.
3. After step 7, I reach the final station, I followed the signs to Noryangjin Fish Market even though I did not find any Exit 3.

6. Critique for Amal's pseudocode

A short (1 page) discussion of how your classmate's algorithms performed in practice. Include information on what worked well, what the failure modes were, and what improvements you would make.

"On paper," the algorithm is well-described, and the sequence of operations in `while` loop makes sense for edge cases:

1. the starting station is not on the optimal line, we transfer immediately
2. at each station that has transfer, we alight the train to check the angle before moving on
3. at end station, even though we don't alight when we don't hear

One immediate optimization I see for the "paper" side is to add a final station recognition for alight train function because otherwise it is not clear for how long we should check the `if train speaker says "Interchange"` condition. Assuming that if we reach a station and do not hear anything, the while loop condition will be checked again, then we will exit.

Speaking of the practical side, the algorithm worked for most part of the path and (almost) delivered me to the destination.

Execution mistakes and bugs that I noticed:

1. When I arrived at Seoul Station, there were 2 entrances to the light and dark blue lines, and there was no nearest subway map to check the angles before I enter, so I made a random decision to go to the light blue line
2. There was no exit 3 when I arrived at the Noryangjin market station. Amal could include a condition that if it does not exist, then the robot can just find the closest exit
3. When I arrived, according to `follow_signs()`, I should first tap out and then follow signs, but the tap out place was at the exit, so I was following the signs to the exit first
4. When I tapped out at Noryangjin and was about the exit, I went into a combination of walls that made me enter an infinite loop following the `walk()` function. We could avoid this by making a termination condition that after several iterations, if we keep turning that try to turn left when there are two choices.

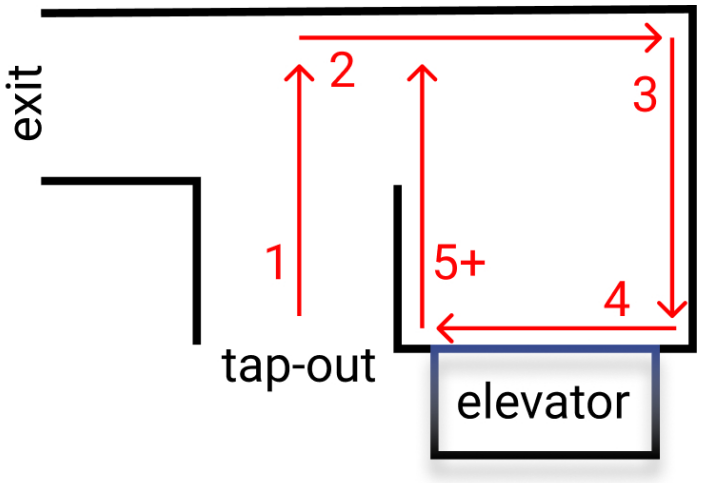


Figure 1. Abstract top-view of the situation at the Noryangjin station exit. This figure demonstrates a room configuration that causes infinite walking loop.

Improvements that could be added:

1. Obstacle dodge. This algorithm dodges only walls but does not take into account people that walk at you or situations when you're walking along a station platform, and there is a pillar that leads you to go on the rails (unsafe). In these cases, we would just need to walk around and switch walking direction.

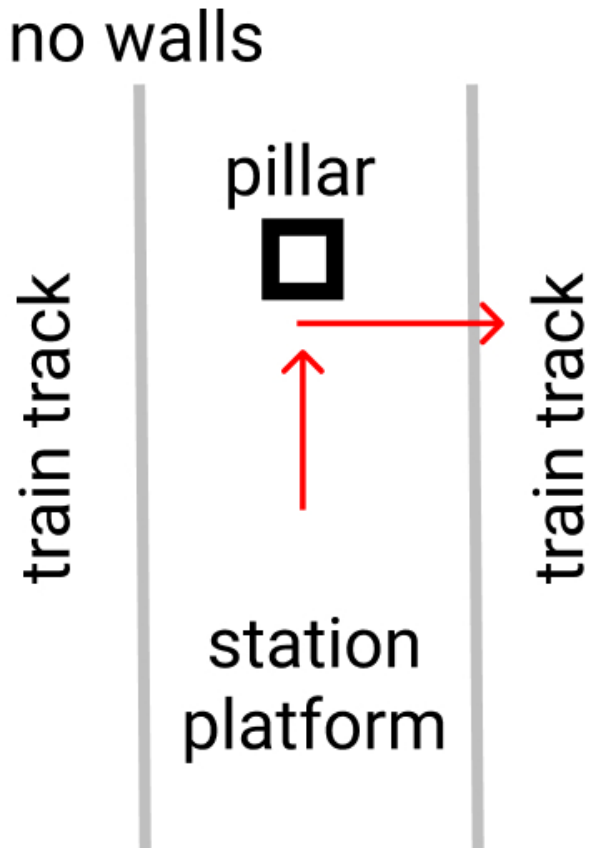


Figure 2. Abstract top-view of a station platform. This figure demonstrates a situation when the walking algorithm would lead to unsafe decisions.

2. As mentioned before, in the `alight_train()` function, a condition check for final station could be added to not let the robot wait until it arrives at the station to realize that it should exit.
3. If this assignment was not to design a greedy algorithm, Amal could employ breadth-first search algorithm from the starting point and stop as soon as it reaches the destination station. Then, the number of layers that we needed would show the minimum distance from between the stations, and, using backtracking, we can find which stations we went through.

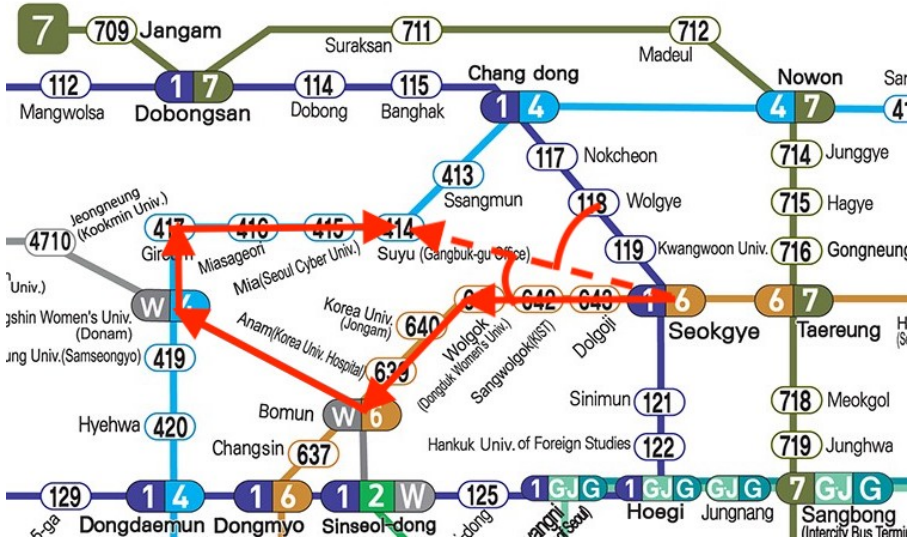


Figure 3. Let's assume that we start at Seokgye station and would like to arrive at Suyu station. Using this greedy algorithm, we could choose the dark yellow path and take 11 stations in total, whereas the global optimal solution is to take the blue dark blue line (6 stations).

Appendix

#simulation: I modeled scenarios where the algorithm would exhibit unpredicted behavior (inifinite feedback loop or falling on the rail track) and explained how they could be solved.
#multipleagents: I described unexpected scenarios that might be caused by other humans (robots) or external objects interracting in the system, such as someone walking at the robot, and how to avoid such situations.
#algorithms: I proposed algorithms to avoid infinite walking feedback loop and to find more optimal path than by greedy algorithm.