MAPPING & NAVIGATION

ICT2104 GROUP B1

Muhammad Ismael Bin Osman

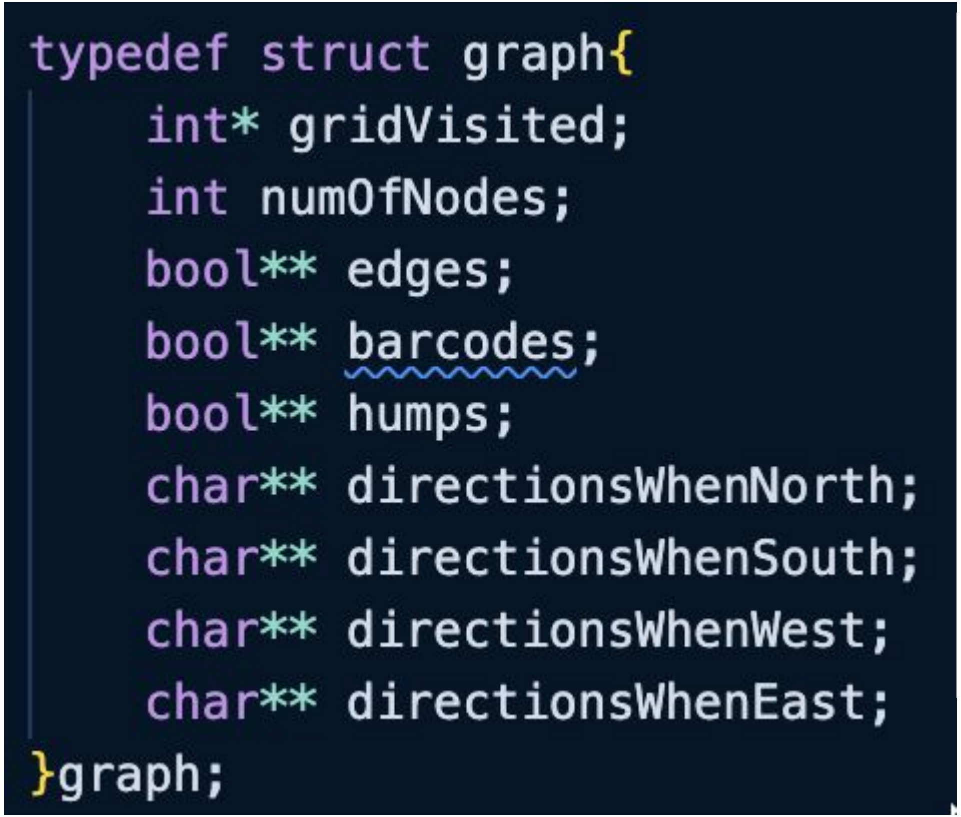
Toh Zheng Hui

Yong Zhi Wei

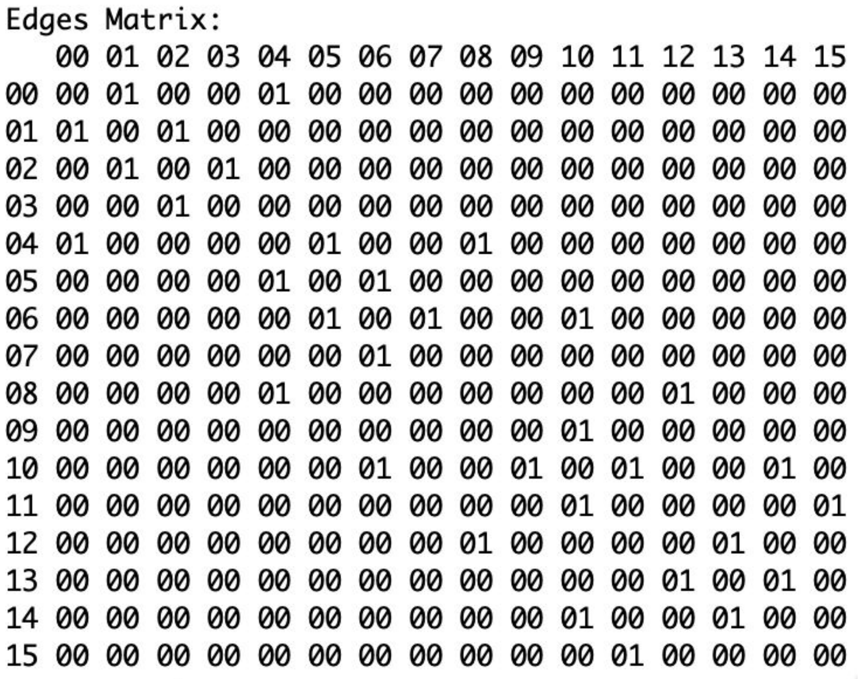
Venkatanarayanan Vishwapriya

MAPPING

Adjacency Matrix

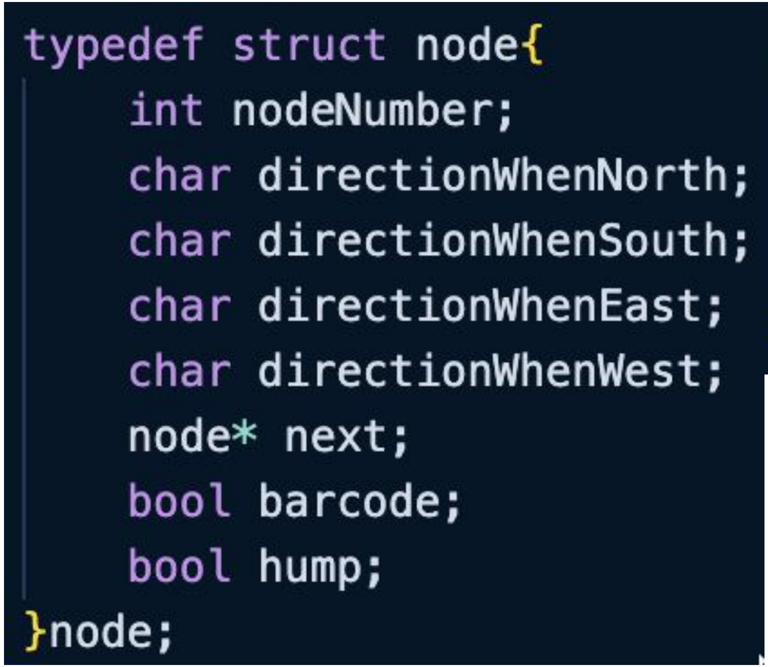


```
typedef struct graph{
    int* gridVisited;
    int numOfNodes;
    bool** edges;
    bool** barcodes;
    bool** humps;
    char** directionsWhenNorth;
    char** directionsWhenSouth;
    char** directionsWhenWest;
    char** directionsWhenEast;
}graph;
```

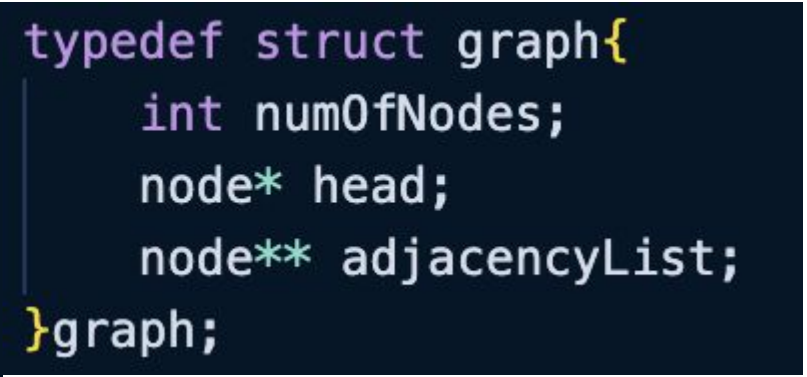


```
Edges Matrix:
   00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
02 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
03 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
04 01 00 00 00 00 01 00 00 01 00 00 00 00 00 00 00
05 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00
06 00 00 00 00 00 01 00 01 00 00 01 00 00 00 00 00
07 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00
08 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00
09 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00
10 00 00 00 00 00 00 01 00 00 01 00 01 00 00 01 00
11 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 01
12 00 00 00 00 00 00 00 00 01 00 00 00 00 01 00 00
13 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 00
14 00 00 00 00 00 00 00 00 00 00 01 00 00 01 00 00
15 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00
```

Adjacency List

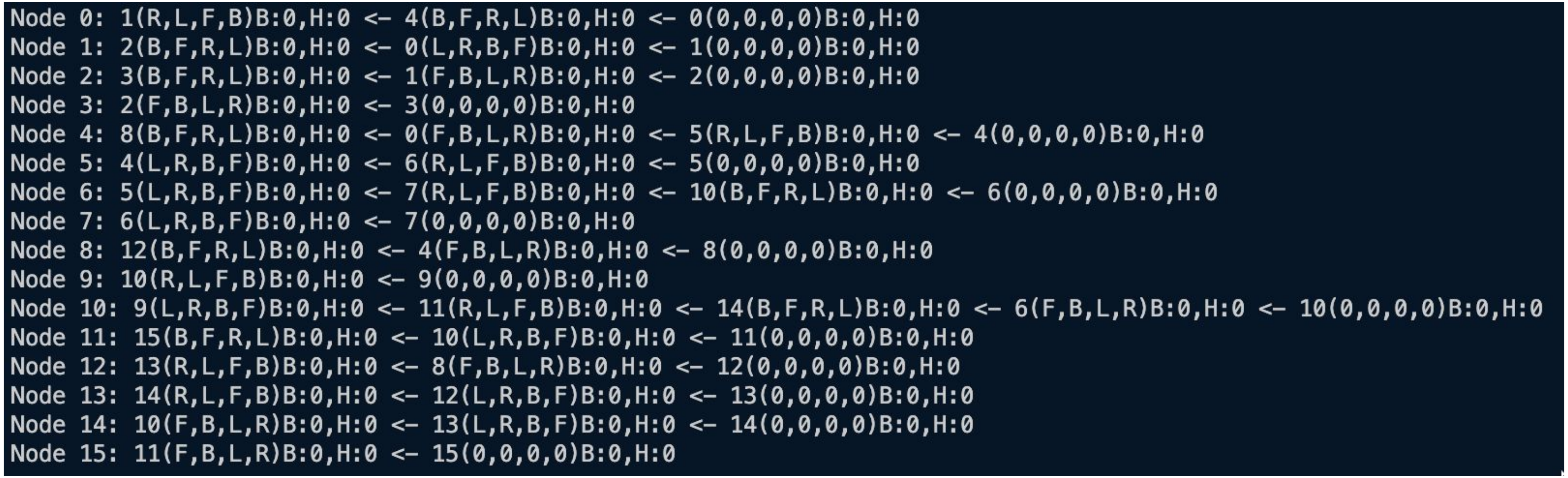


```
typedef struct node{
    int nodeNumber;
    char directionWhenNorth;
    char directionWhenSouth;
    char directionWhenEast;
    char directionWhenWest;
    node* next;
    bool barcode;
    bool hump;
}node;
```



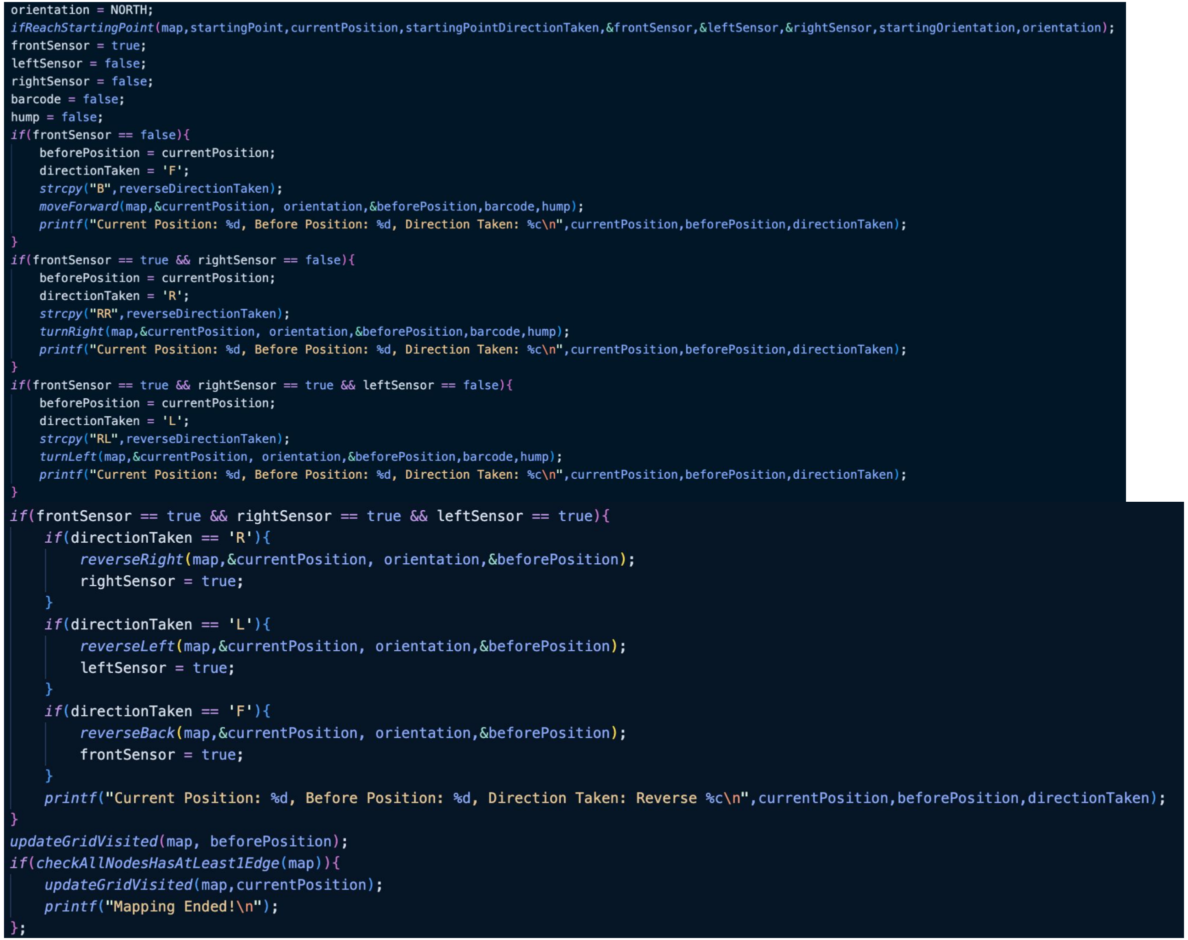
```
typedef struct graph{
    int numOfNodes;
    node* head;
    node** adjacencyList;
}graph;
```

Adjacency List



```
Node 0: 1(R,L,F,B)B:0,H:0 <- 4(B,F,R,L)B:0,H:0 <- 0(0,0,0,0)B:0,H:0
Node 1: 2(B,F,R,L)B:0,H:0 <- 0(L,R,B,F)B:0,H:0 <- 1(0,0,0,0)B:0,H:0
Node 2: 3(B,F,R,L)B:0,H:0 <- 1(F,B,L,R)B:0,H:0 <- 2(0,0,0,0)B:0,H:0
Node 3: 2(F,B,L,R)B:0,H:0 <- 3(0,0,0,0)B:0,H:0
Node 4: 8(B,F,R,L)B:0,H:0 <- 0(F,B,L,R)B:0,H:0 <- 5(R,L,F,B)B:0,H:0 <- 4(0,0,0,0)B:0,H:0
Node 5: 4(L,R,B,F)B:0,H:0 <- 6(R,L,F,B)B:0,H:0 <- 5(0,0,0,0)B:0,H:0
Node 6: 5(L,R,B,F)B:0,H:0 <- 7(R,L,F,B)B:0,H:0 <- 10(B,F,R,L)B:0,H:0 <- 6(0,0,0,0)B:0,H:0
Node 7: 6(L,R,B,F)B:0,H:0 <- 7(0,0,0,0)B:0,H:0
Node 8: 12(B,F,R,L)B:0,H:0 <- 4(F,B,L,R)B:0,H:0 <- 8(0,0,0,0)B:0,H:0
Node 9: 10(R,L,F,B)B:0,H:0 <- 9(0,0,0,0)B:0,H:0
Node 10: 9(L,R,B,F)B:0,H:0 <- 11(R,L,F,B)B:0,H:0 <- 14(B,F,R,L)B:0,H:0 <- 6(F,B,L,R)B:0,H:0 <- 10(0,0,0,0)B:0,H:0
Node 11: 15(B,F,R,L)B:0,H:0 <- 10(L,R,B,F)B:0,H:0 <- 11(0,0,0,0)B:0,H:0
Node 12: 13(R,L,F,B)B:0,H:0 <- 8(F,B,L,R)B:0,H:0 <- 12(0,0,0,0)B:0,H:0
Node 13: 14(R,L,F,B)B:0,H:0 <- 12(L,R,B,F)B:0,H:0 <- 13(0,0,0,0)B:0,H:0
Node 14: 10(F,B,L,R)B:0,H:0 <- 13(L,R,B,F)B:0,H:0 <- 14(0,0,0,0)B:0,H:0
Node 15: 11(F,B,L,R)B:0,H:0 <- 15(0,0,0,0)B:0,H:0
```

General Flow



```c
orientation = NORTH;
ifReachStartingPoint(map,startingPoint,currentPosition,startingPointDirectionTaken,&frontSensor,&leftSensor,&rightSensor,startingOrientation,orientation);
frontSensor = true;
leftSensor = false;
rightSensor = false;
barcode = false;
hump = false;
if(frontSensor == false){
    beforePosition = currentPosition;
    directionTaken = 'F';
    strcpy("B",reverseDirectionTaken);
    moveForward(map,&currentPosition, orientation,&beforePosition,barcode,hump);
    printf("Current Position: %d, Before Position: %d, Direction Taken: %c\n",currentPosition,beforePosition,directionTaken);
}
if(frontSensor == true && rightSensor == false){
    beforePosition = currentPosition;
    directionTaken = 'R';
    strcpy("RR",reverseDirectionTaken);
    turnRight(map,&currentPosition, orientation,&beforePosition,barcode,hump);
    printf("Current Position: %d, Before Position: %d, Direction Taken: %c\n",currentPosition,beforePosition,directionTaken);
}
if(frontSensor == true && rightSensor == true && leftSensor == false){
    beforePosition = currentPosition;
    directionTaken = 'L';
    strcpy("RL",reverseDirectionTaken);
    turnLeft(map,&currentPosition, orientation,&beforePosition,barcode,hump);
    printf("Current Position: %d, Before Position: %d, Direction Taken: %c\n",currentPosition,beforePosition,directionTaken);
}
if(frontSensor == true && rightSensor == true && leftSensor == true){
    if(directionTaken == 'R'){
        reverseRight(map,&currentPosition, orientation,&beforePosition);
        rightSensor = true;
    }
    if(directionTaken == 'L'){
        reverseLeft(map,&currentPosition, orientation,&beforePosition);
        leftSensor = true;
    }
    if(directionTaken == 'F'){
        reverseBack(map,&currentPosition, orientation,&beforePosition);
        frontSensor = true;
    }
    printf("Current Position: %d, Before Position: %d, Direction Taken: Reverse %c\n",currentPosition,beforePosition,directionTaken);
}
updateGridVisited(map, beforePosition);
if(checkAllNodesHasAtLeast1Edge(map)){
    updateGridVisited(map,currentPosition);
    printf("Mapping Ended!\n");
};
```

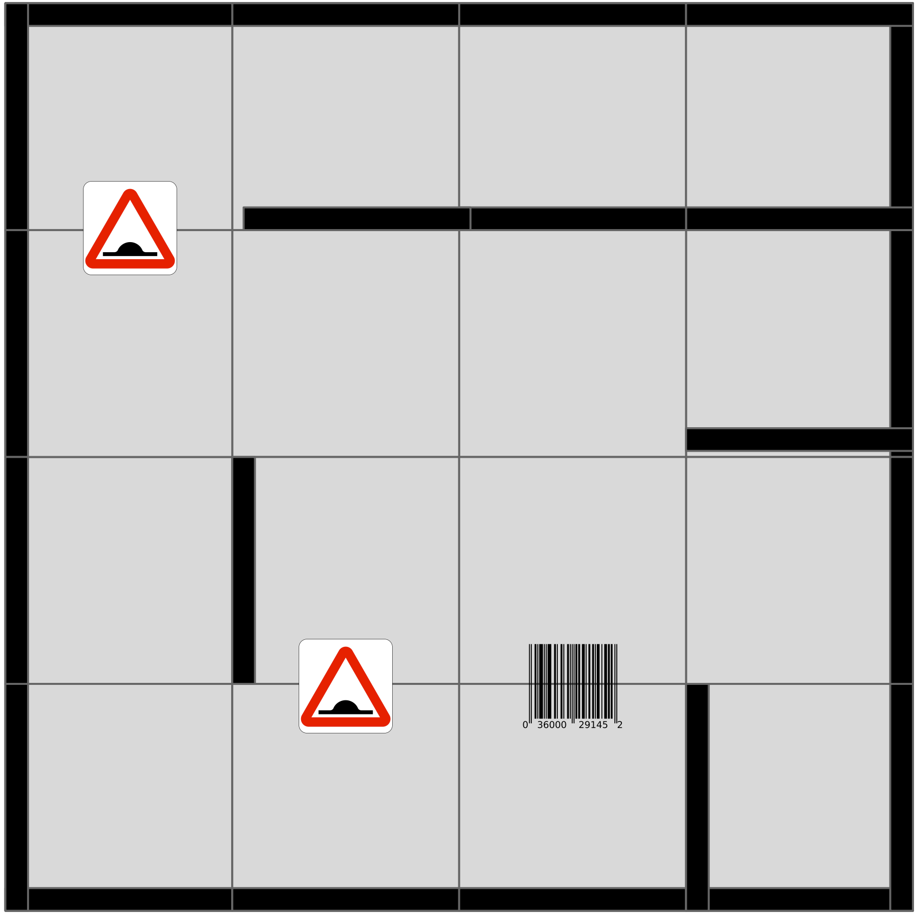
Map Example

Walls

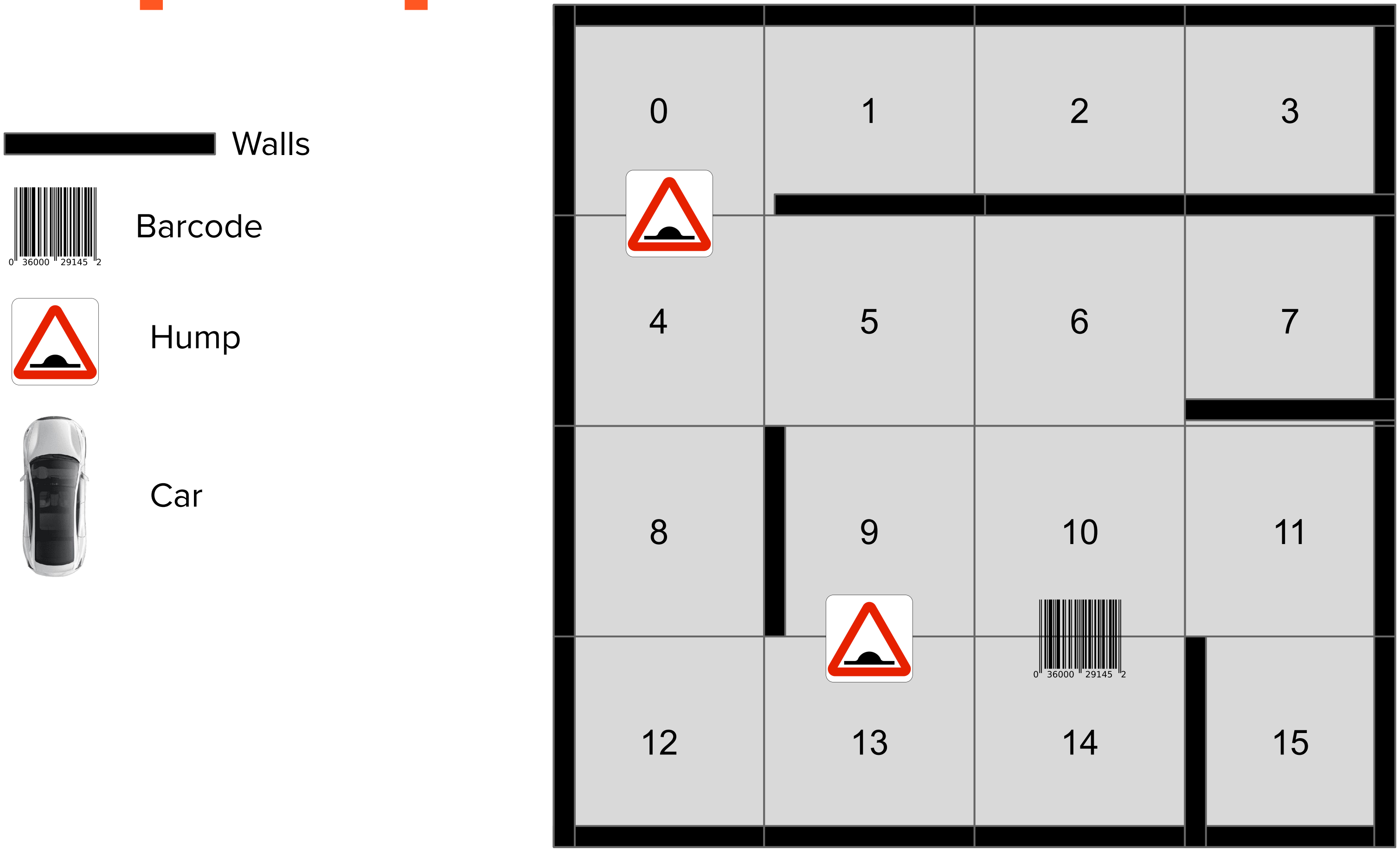
Barcode

Hump

Car



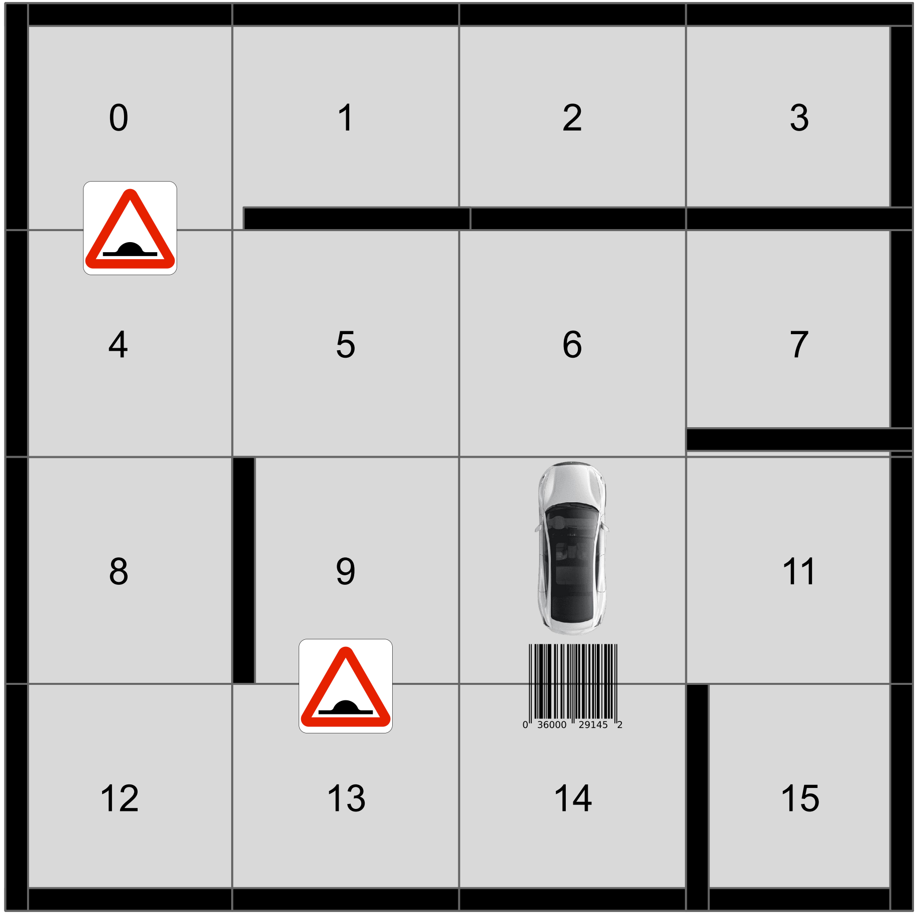
Map Example


Walls
Barcode
0 36000 29145 2
Hump
Car
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
0 36000 29145 2

Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

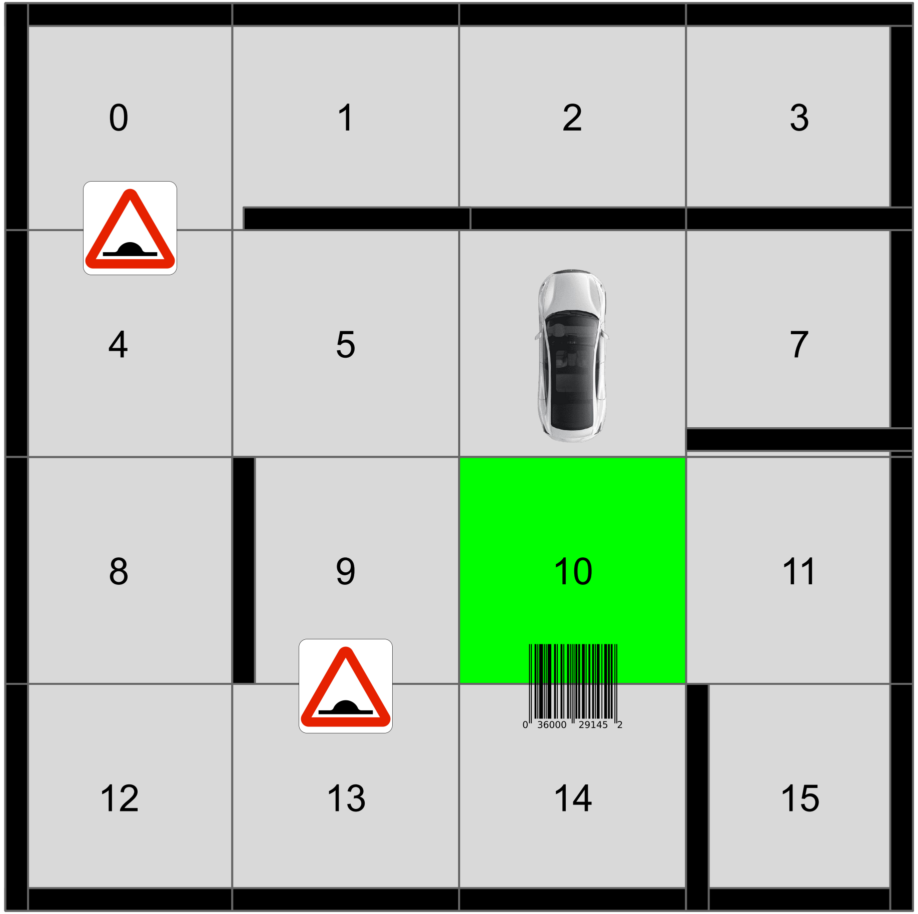
Current Location - 10
Current Orientation - North
Grid Traveled = []



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

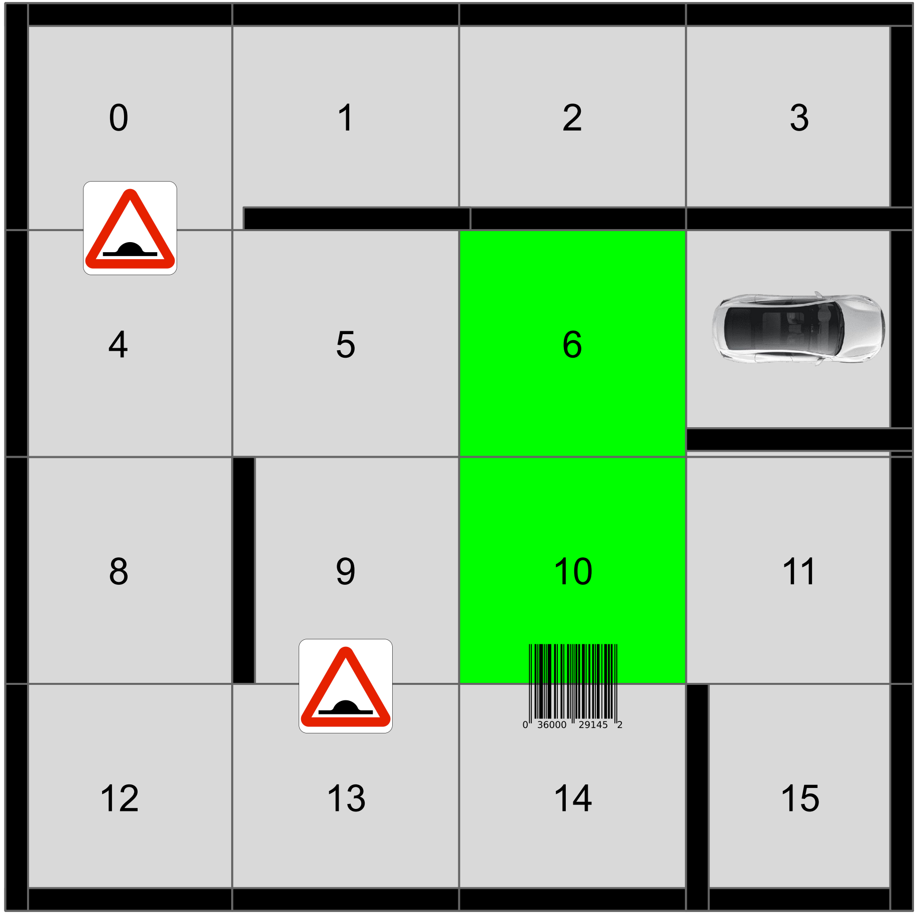
Current Location - 6
Current Orientation - North
Grid Traveled = [10,]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 7
Current Orientation - East
Grid Traveled = [10,6]

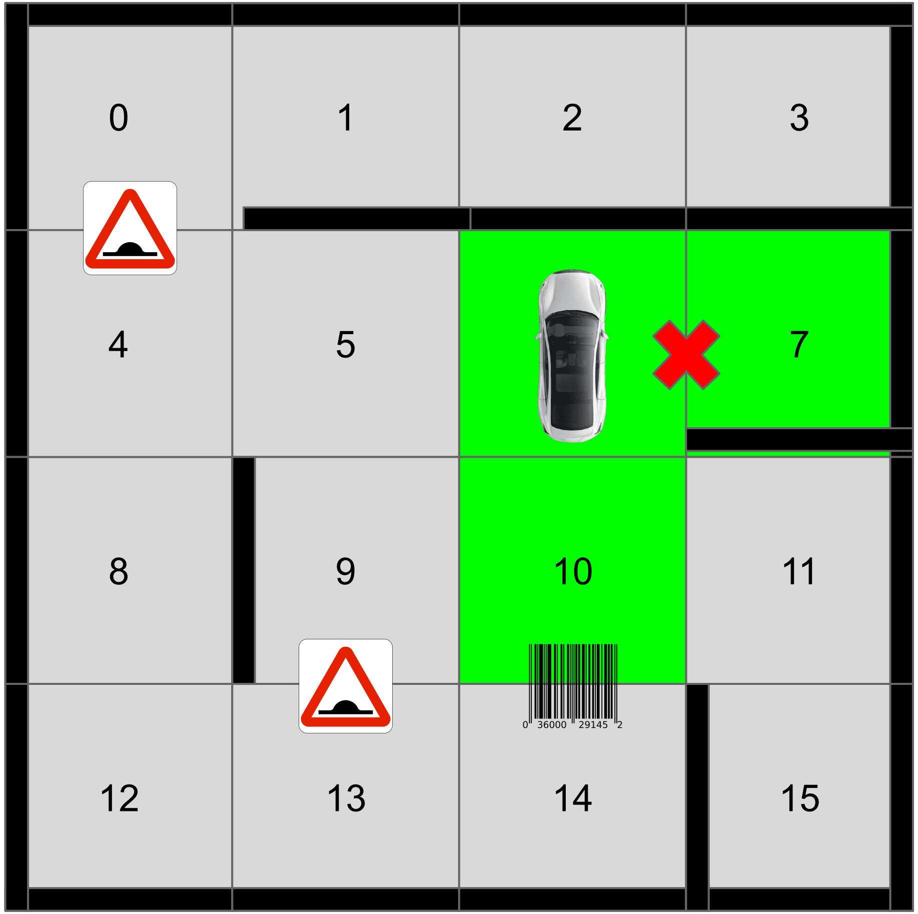


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 6
Current Orientation - East
Grid Traveled = [10,6,7]

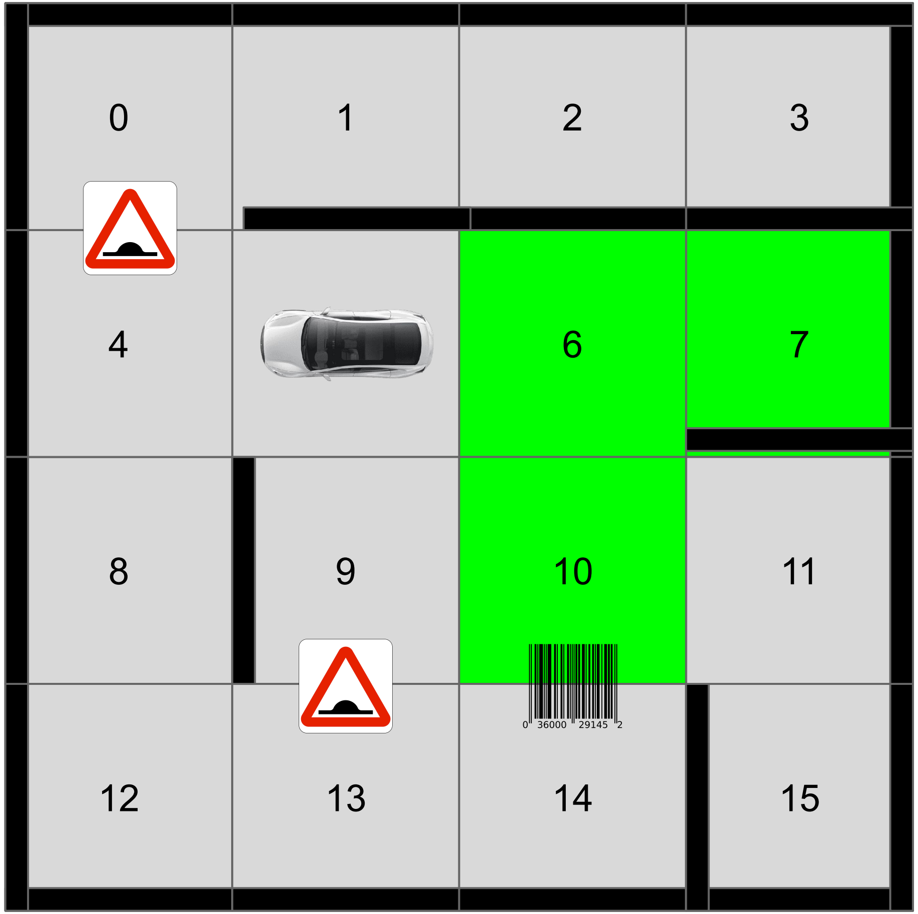
Right Sensor = BLOCKED



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 5
Current Orientation - West
Grid Traveled = [10,6,7]

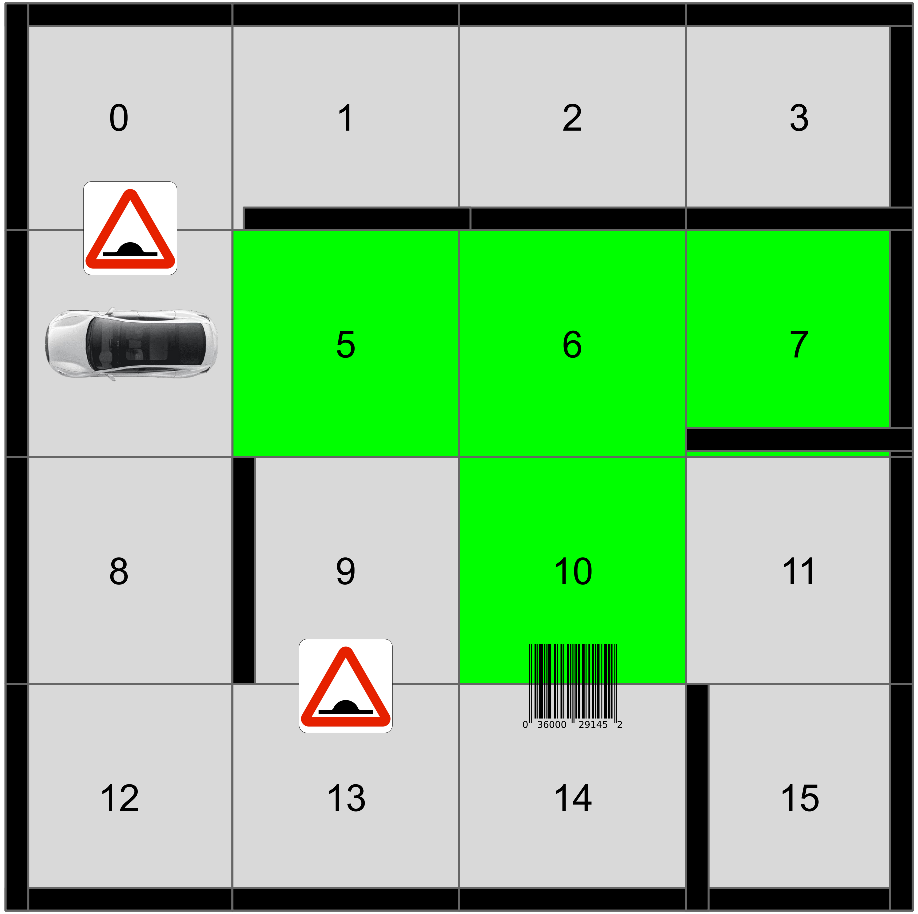


0	1	2	3
4		6	7
8	9	10	11
12	13	14	15

Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 4
Current Orientation - West
Grid Traveled = [10,6,7,5]

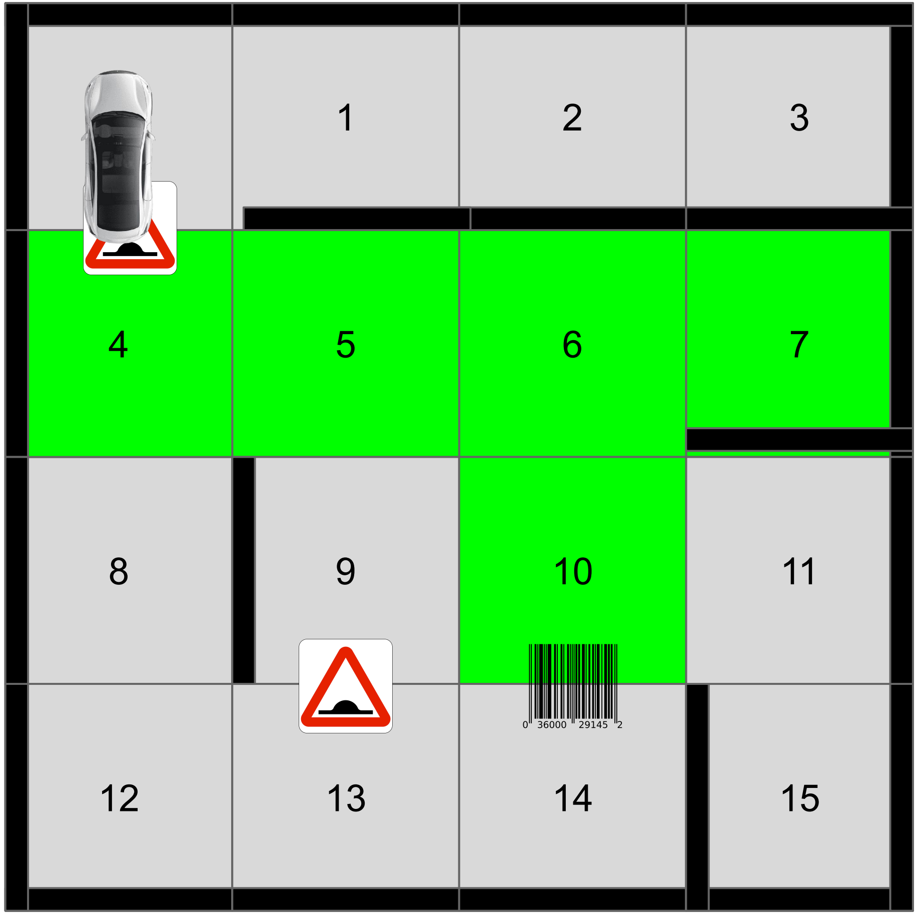


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 0
Current Orientation - North
Grid Traveled = [10,6,7,5,4]

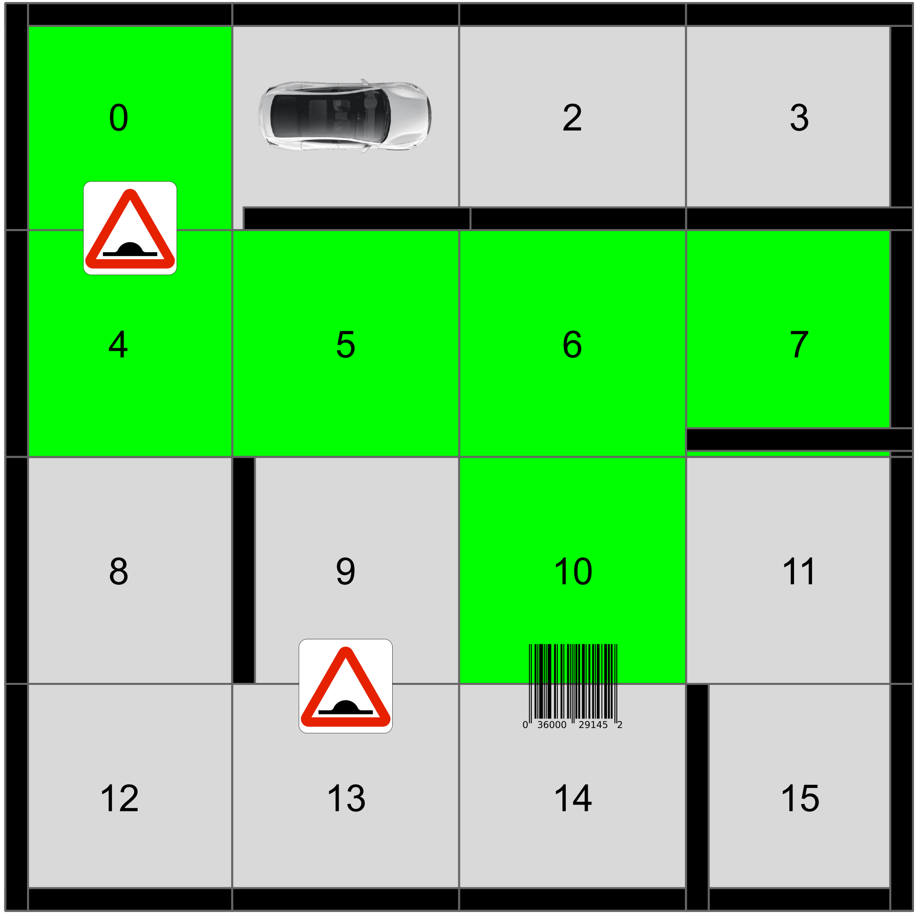
Hump Detected!



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

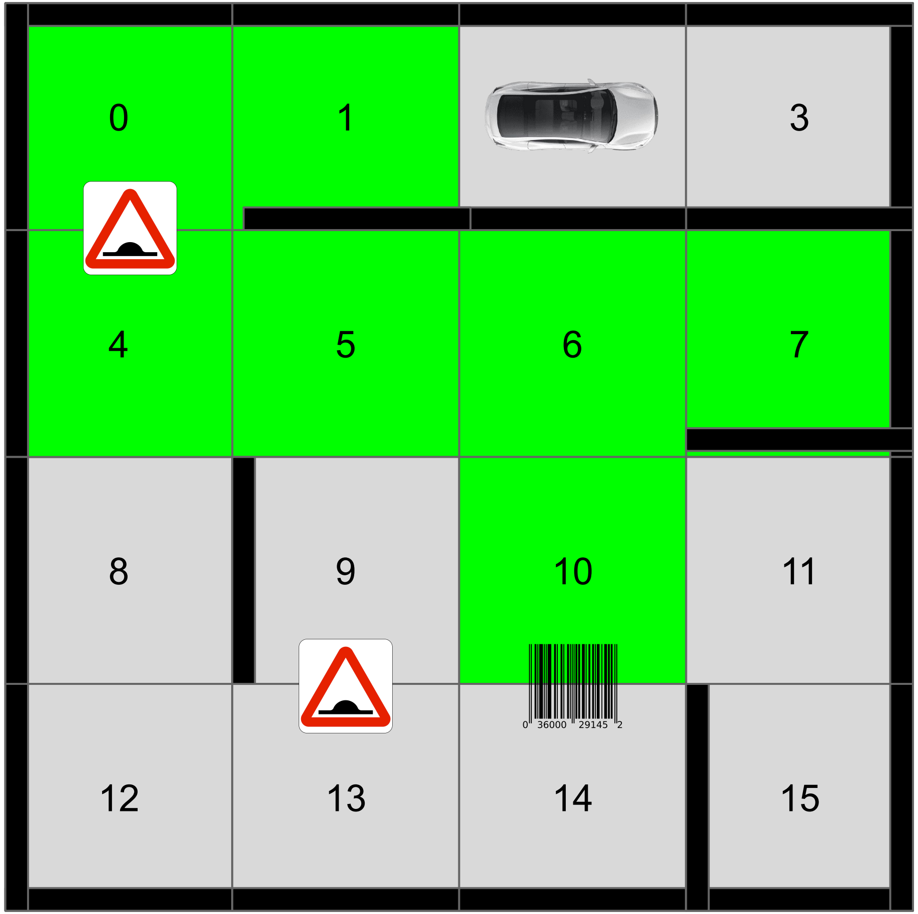
Current Location - 1
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

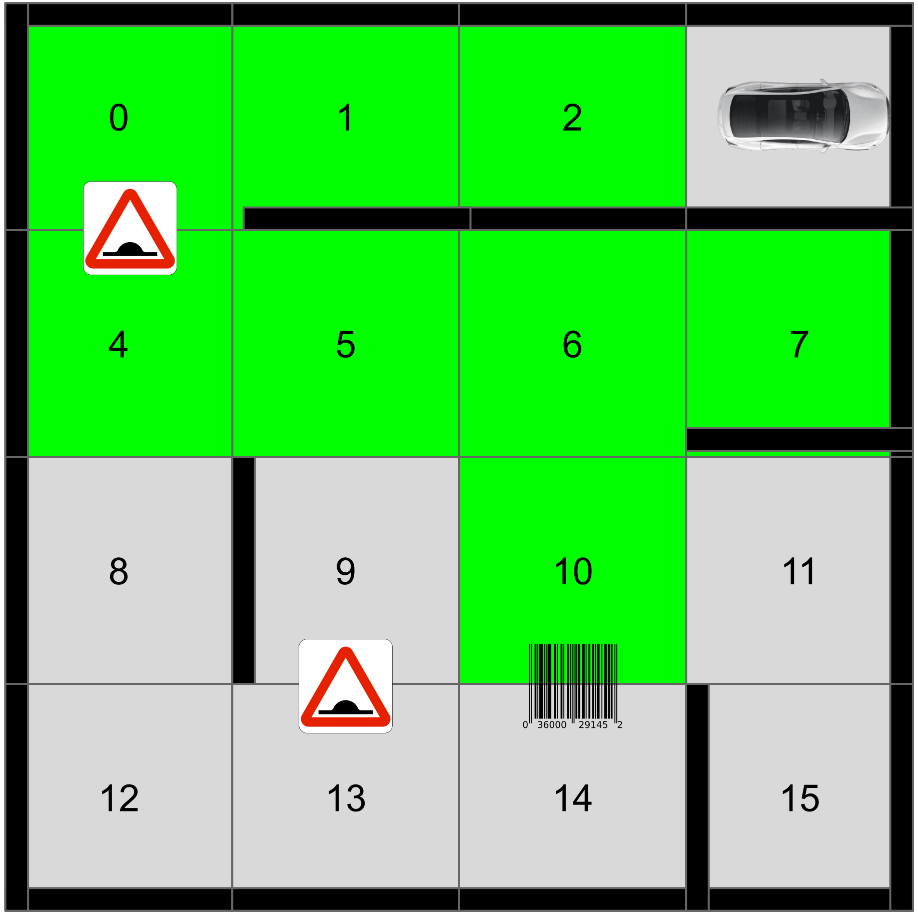
Current Location - 2
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0,1]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 3
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0,1,2]

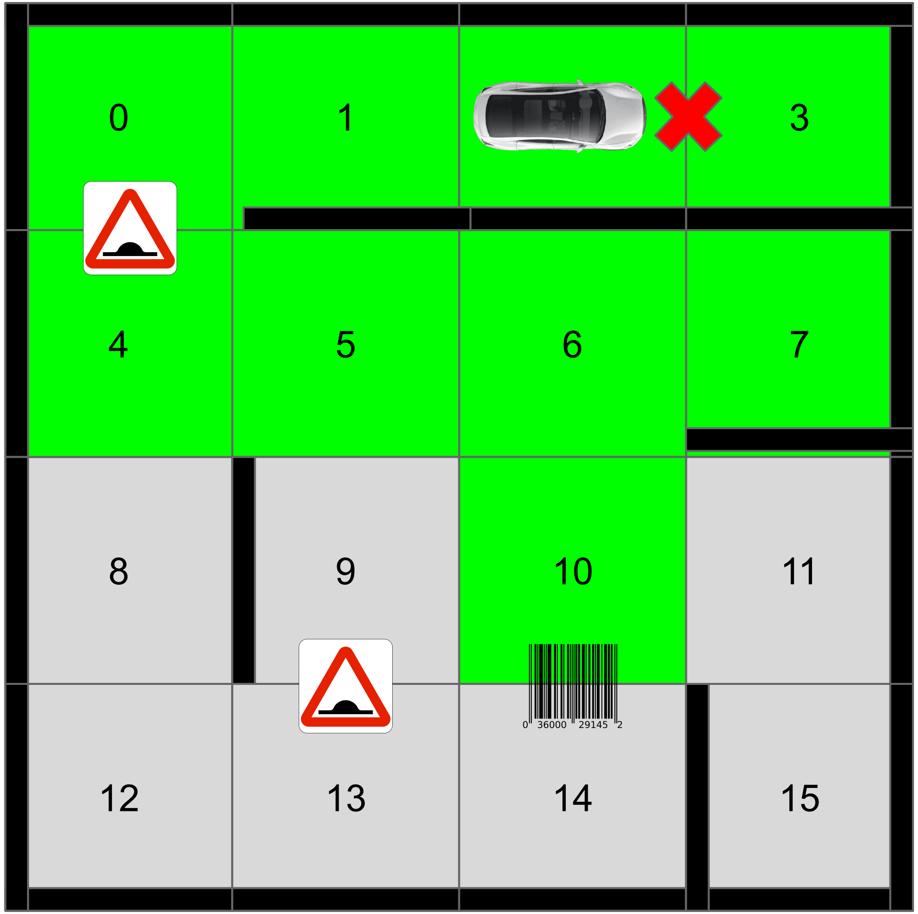


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 2
Current Orientation - West
Grid Traveled = [10,6,7,5,4,0,1,2,3]

Front Sensor = BLOCKED

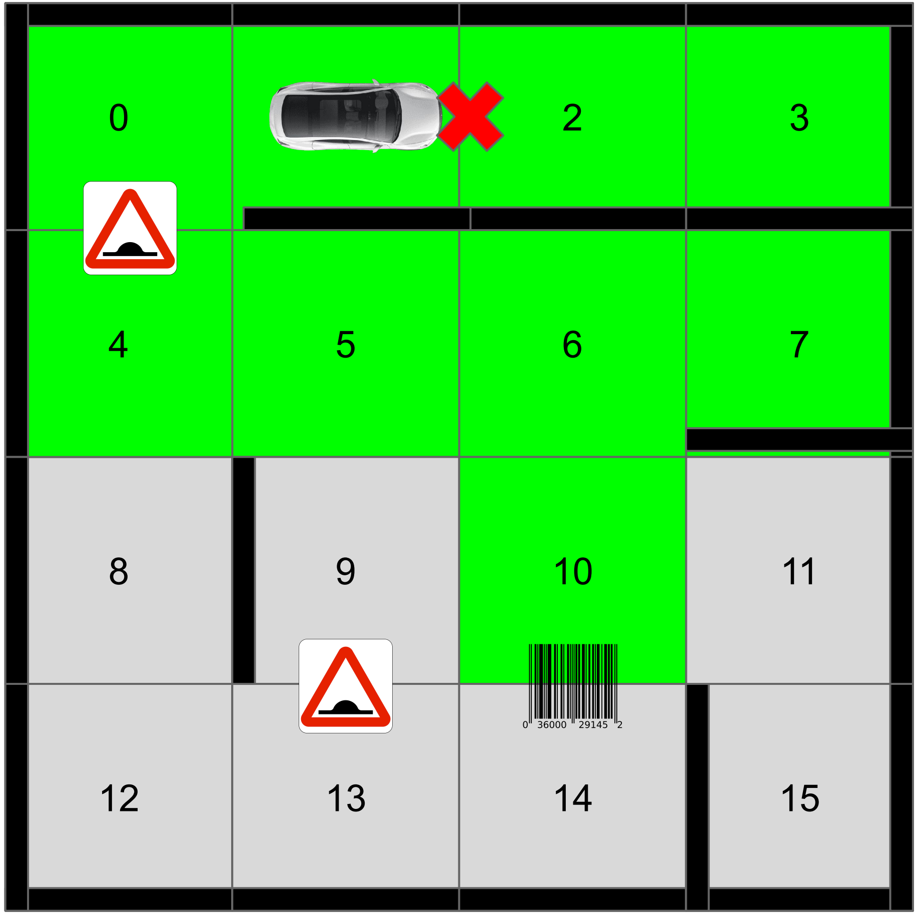


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 1
Current Orientation - West
Grid Traveled = [10,6,7,5,4,0,1,2,3]

Front Sensor = BLOCKED

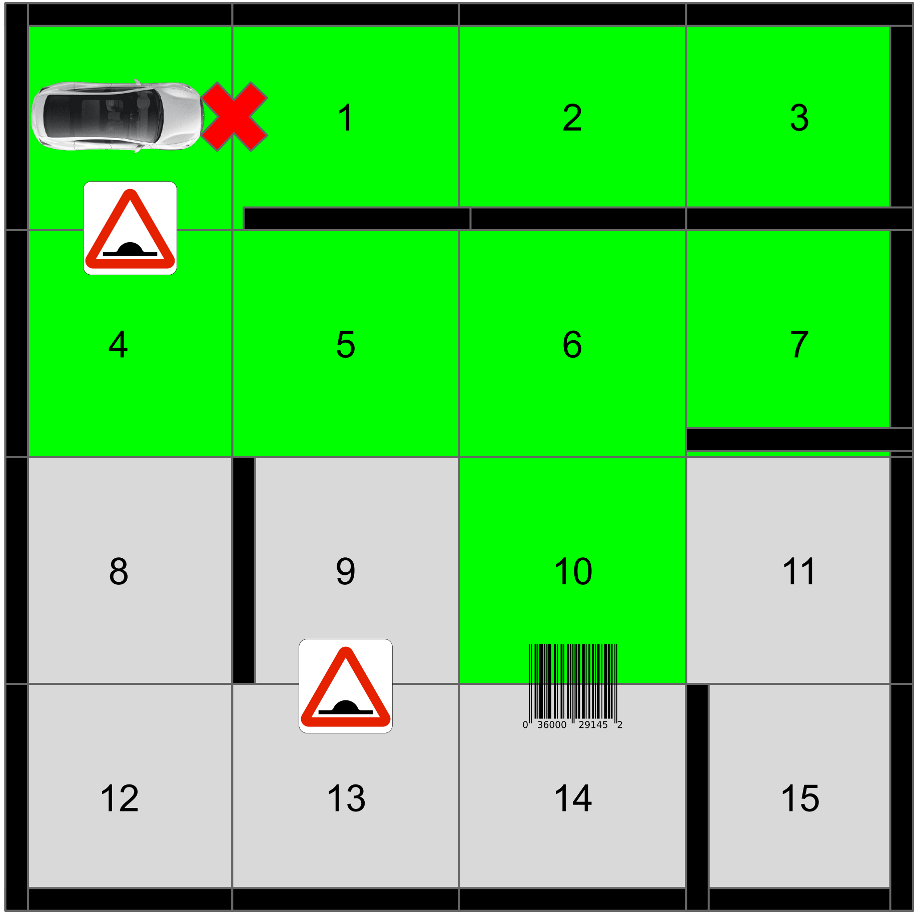


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 0
Current Orientation - West
Grid Traveled =
[10,6,7,5,4,0,1,2,3]

Front Sensor = BLOCKED

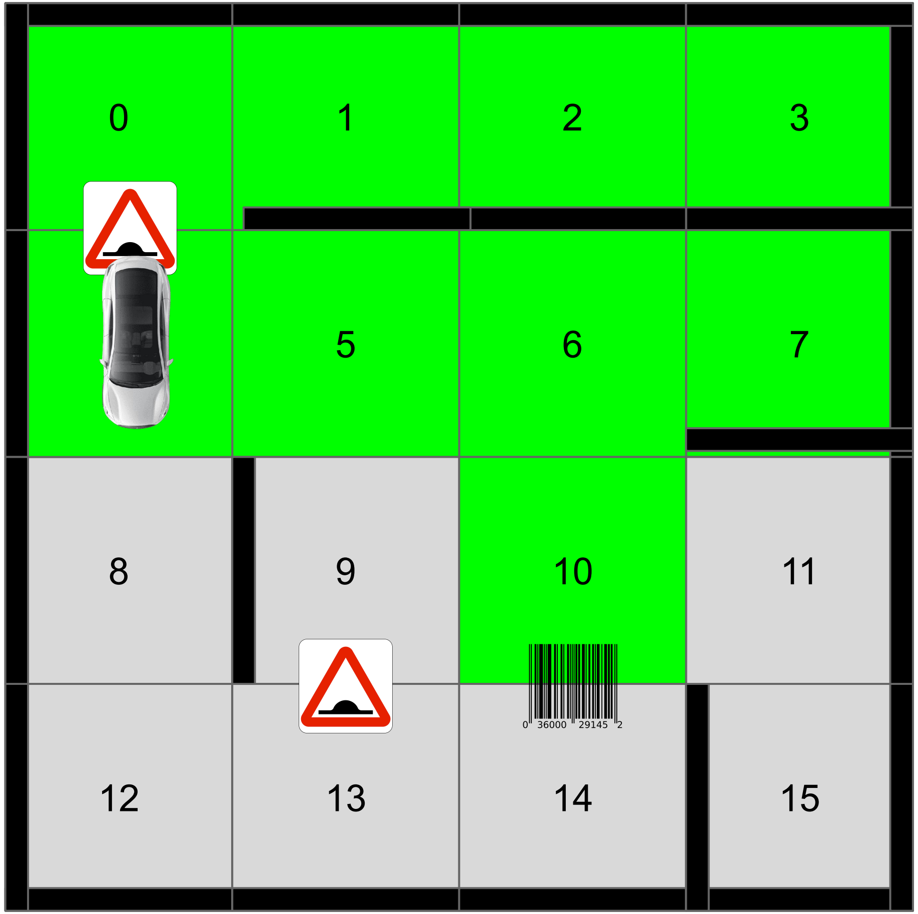


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 4
Current Orientation - South
Grid Traveled = [10,6,7,5,4,0,1,2,3]

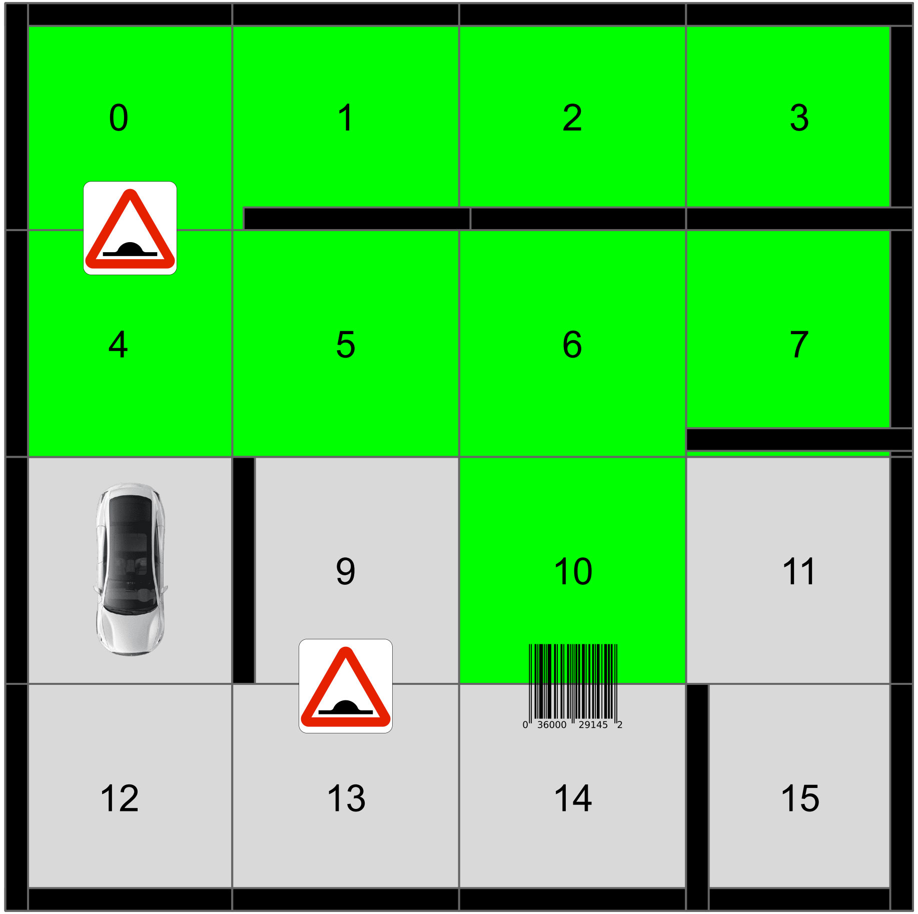
Hump Detected!



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

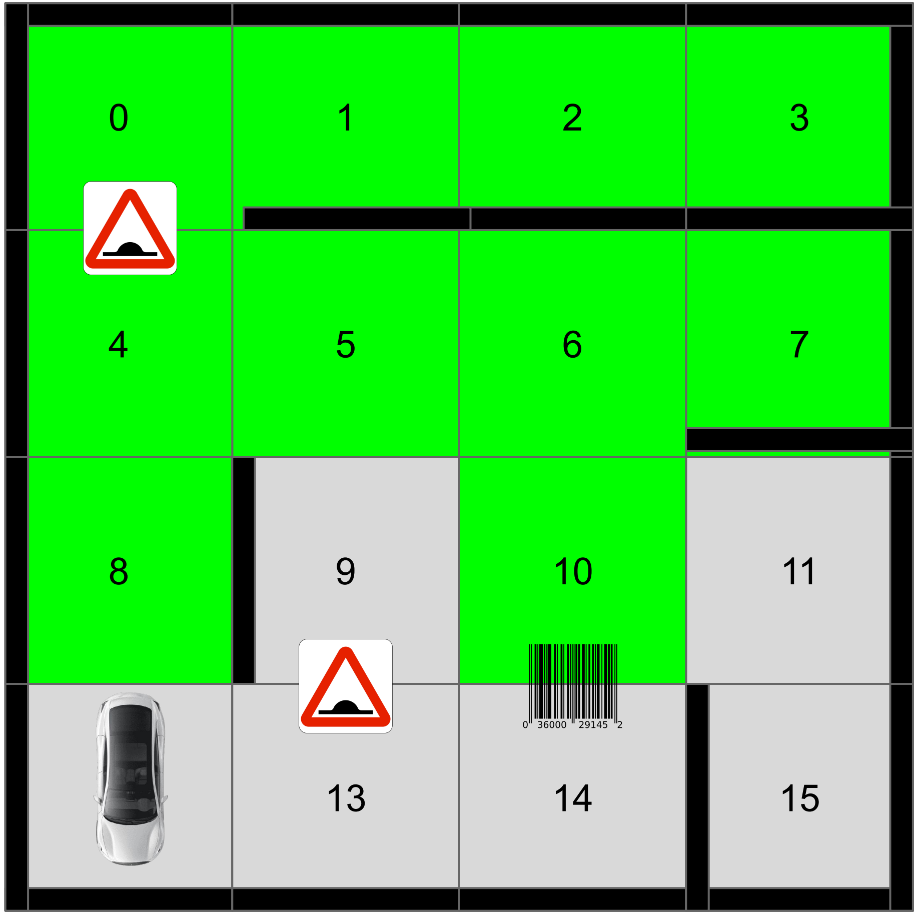
Current Location - 8
Current Orientation - South
Grid Traveled = [10,6,7,5,4,0,1,2,3]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

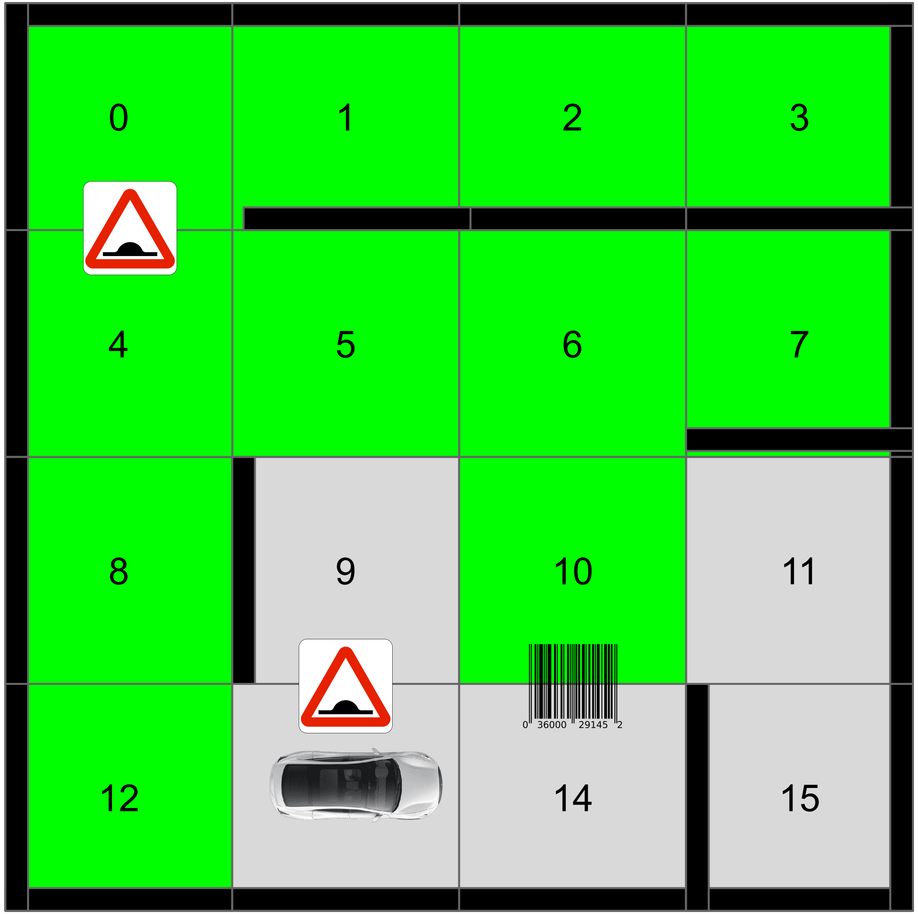
Current Location - 12
Current Orientation - South
Grid Traveled = [10,6,7,5,4,0,1,2,3,8]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

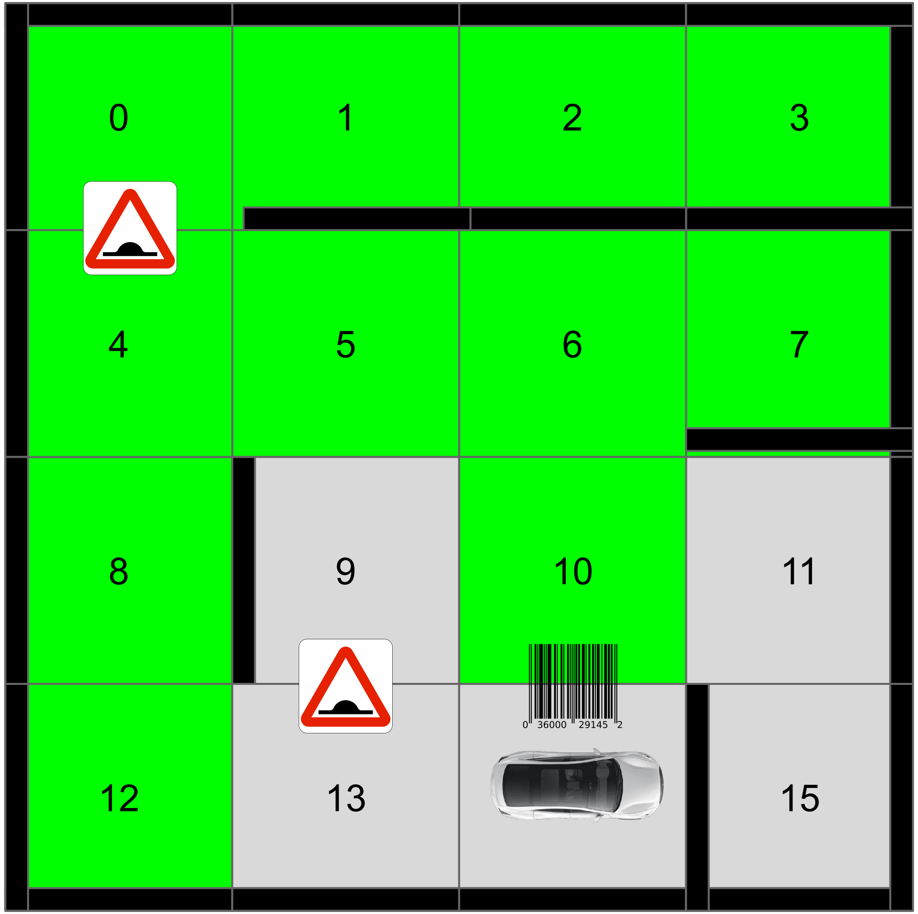
Current Location - 13
Current Orientation - East
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

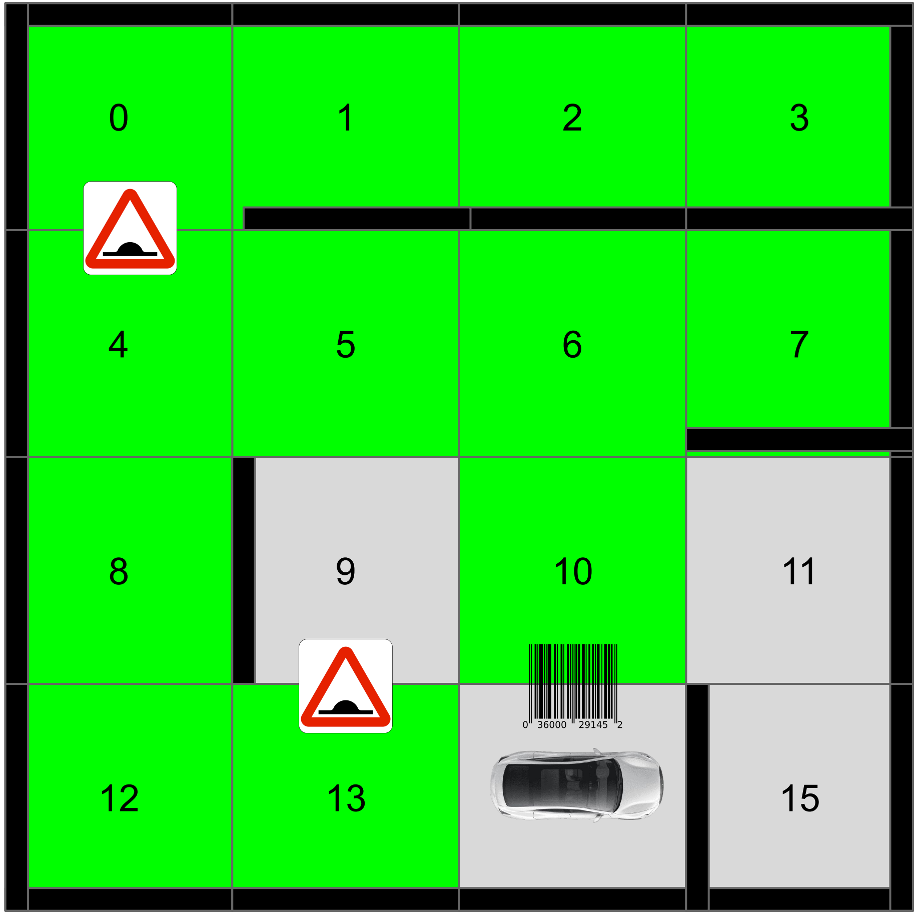
Current Location - 13
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0,1,2,3,8,12]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 14
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0,1,2,3,8,12,13]

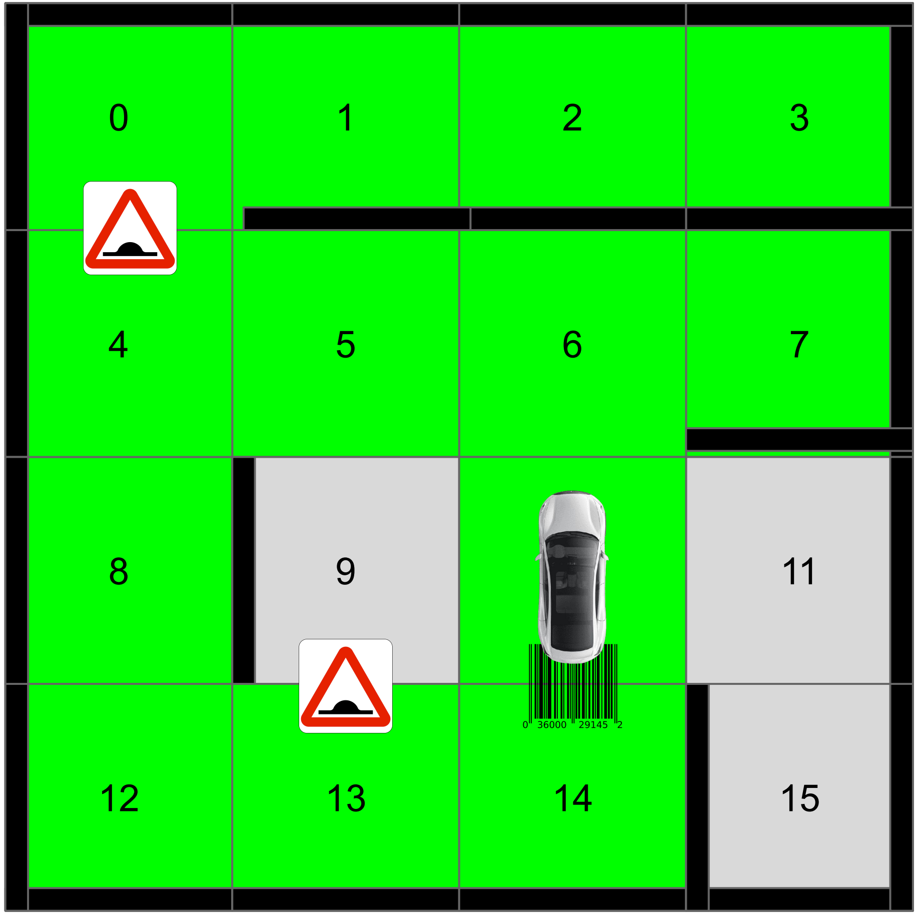


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 10
Current Orientation - North
Grid Traveled = [10,6,7,5,4,0,1,2,3,8,12,13,14]

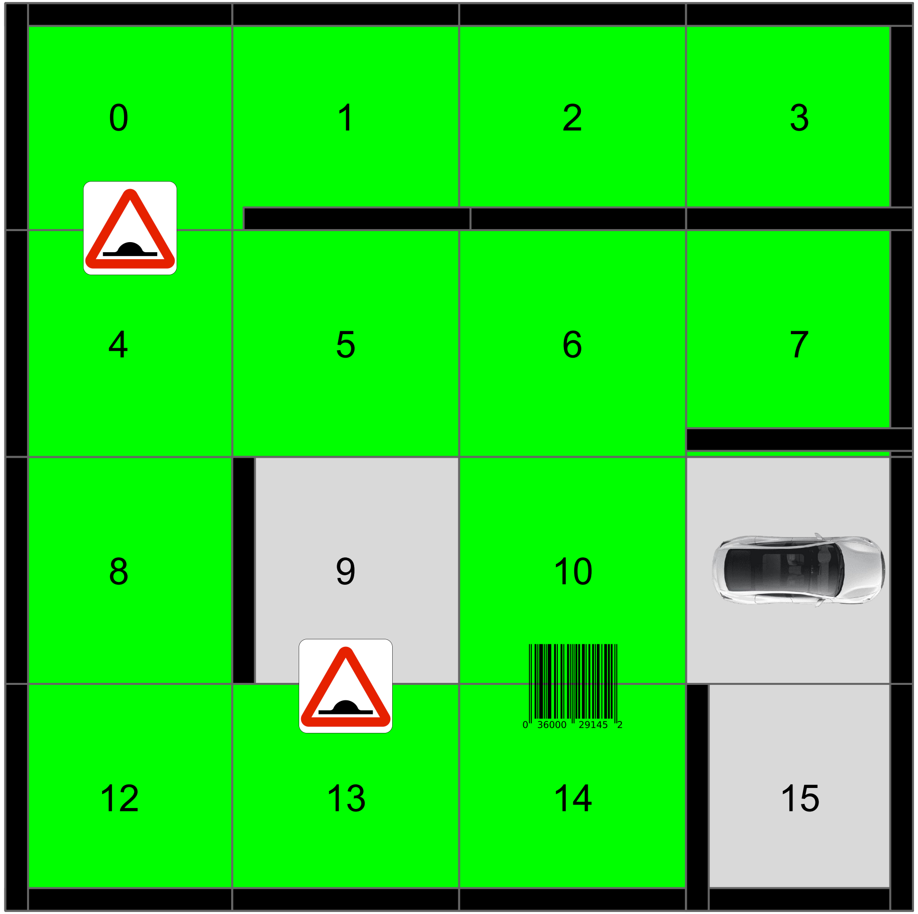
If reach back at starting point, check if already travelled to front grid. If have, turn right



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

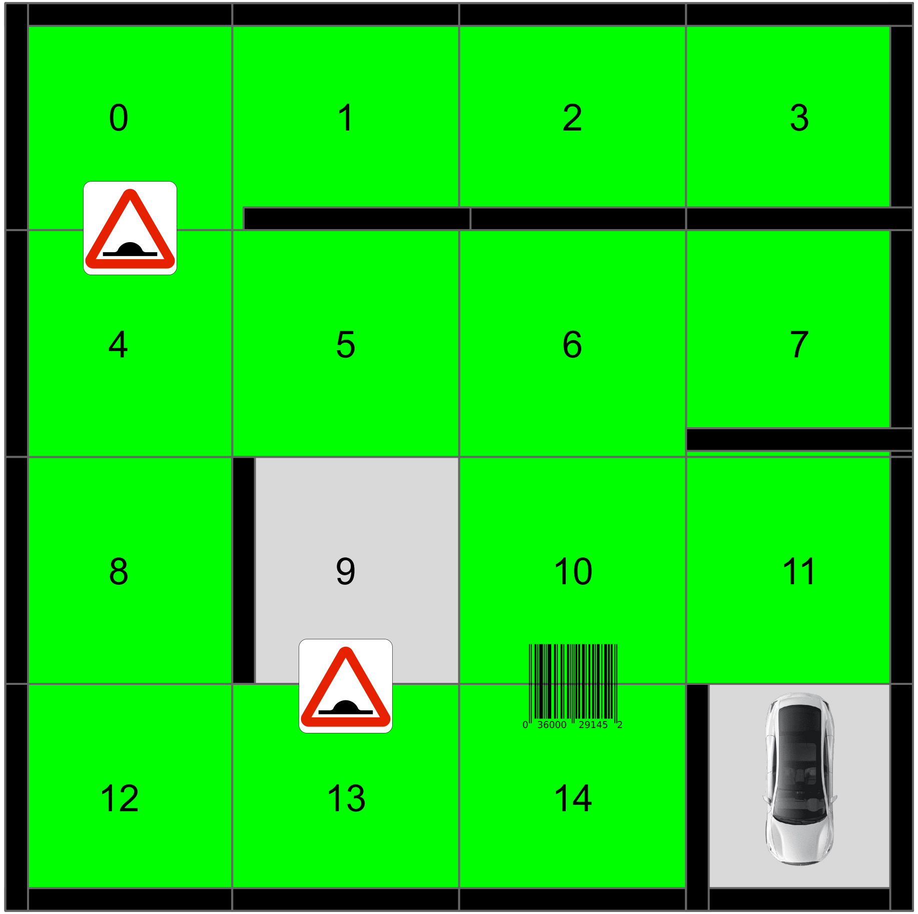
Current Location - 11
Current Orientation - East
Grid Traveled = [10,6,7,5,4,0,1,2,3,8,12,13,14]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

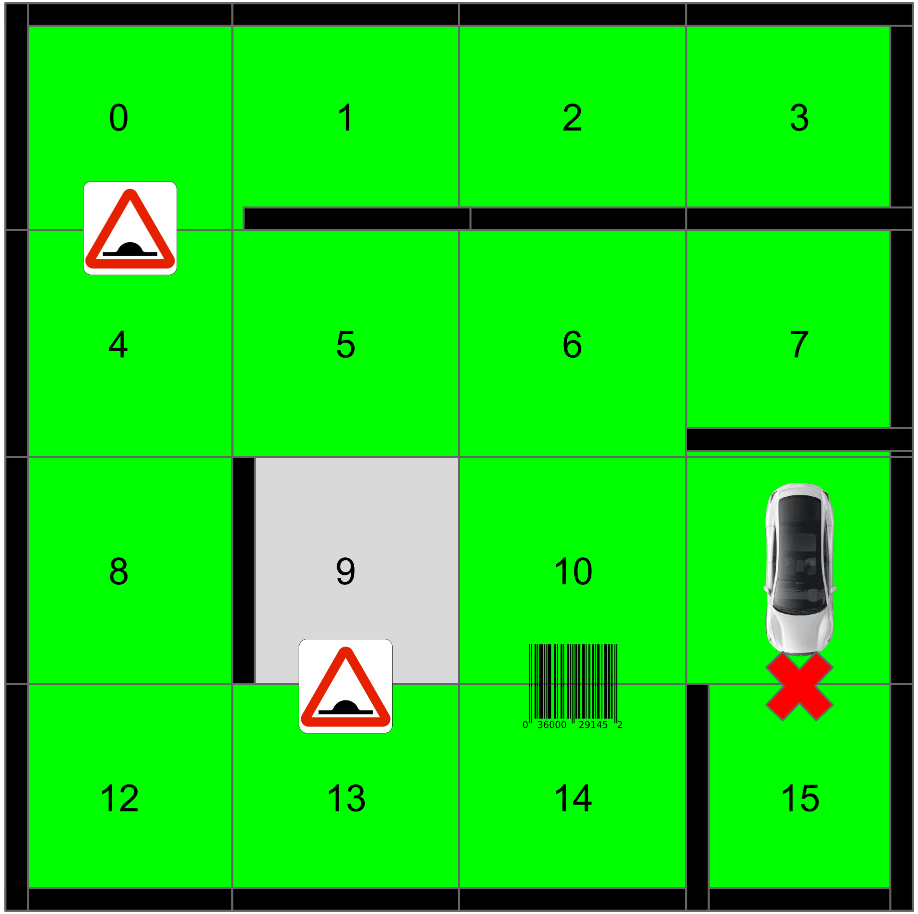
Current Location - 15
Current Orientation - South
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12,13,14,11]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

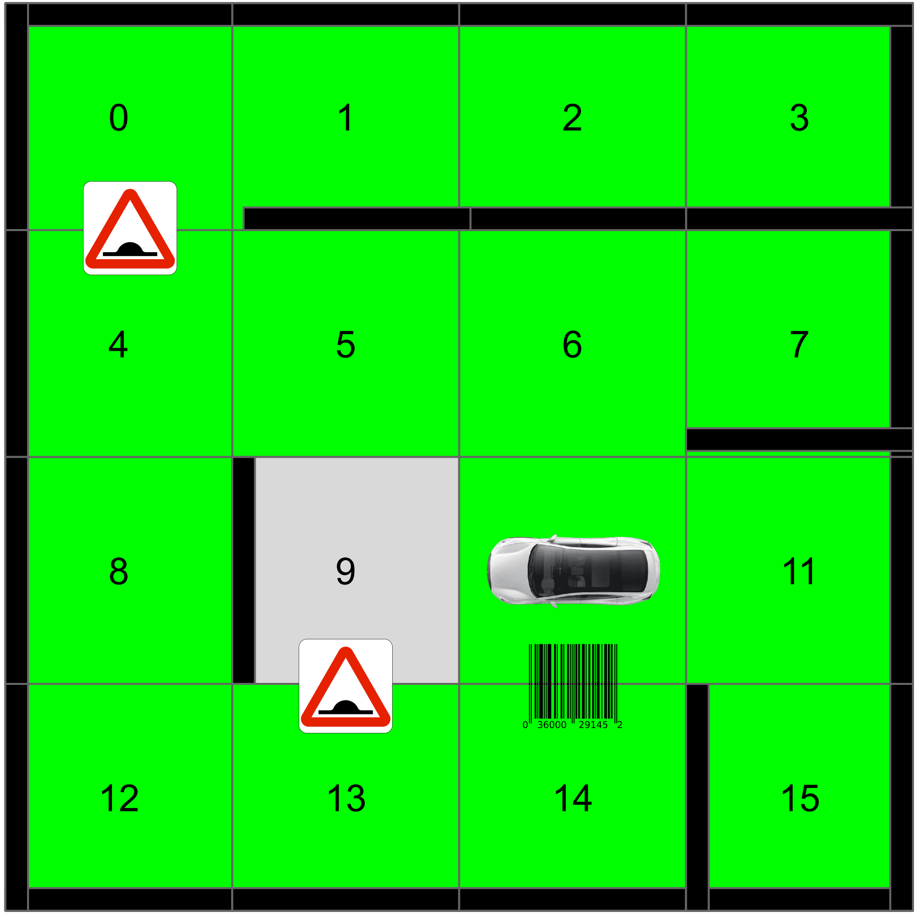
Current Location - 11
Current Orientation - South
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12,13,14,11,15]



Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 10
Current Orientation - West
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12,13,14,11,15]

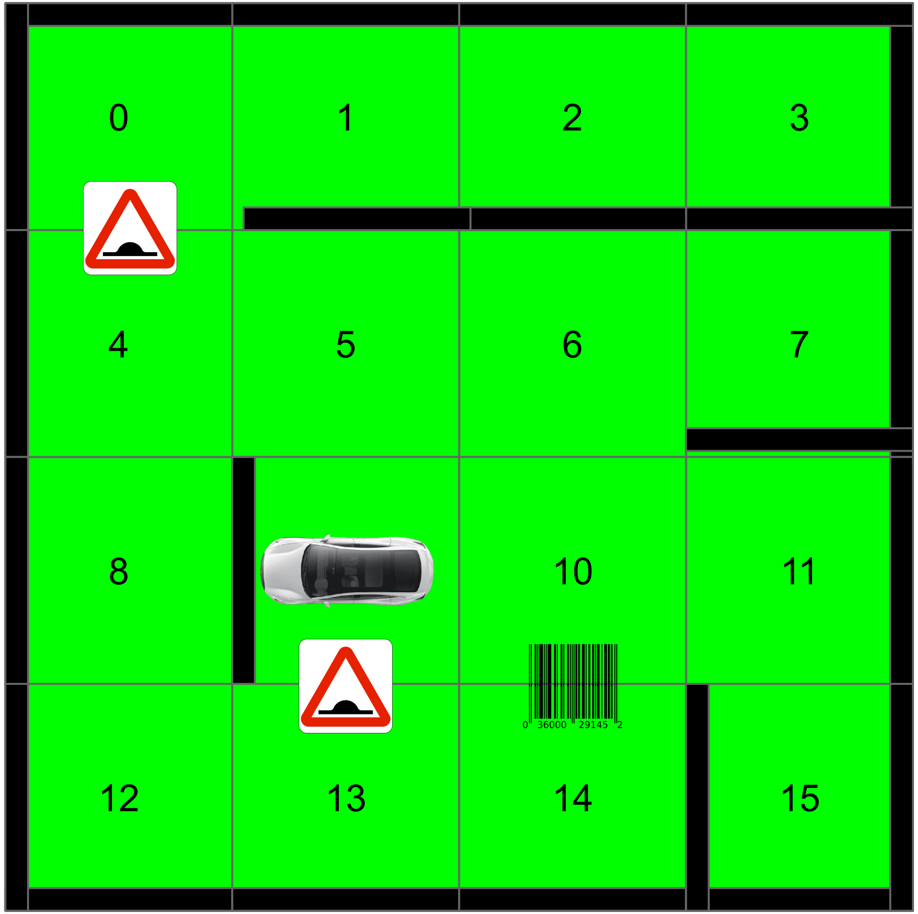


Map Example

Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 9
Current Orientation - West
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12,13,14,11,15,9]

Once all grids has at least one edge, mapping stops

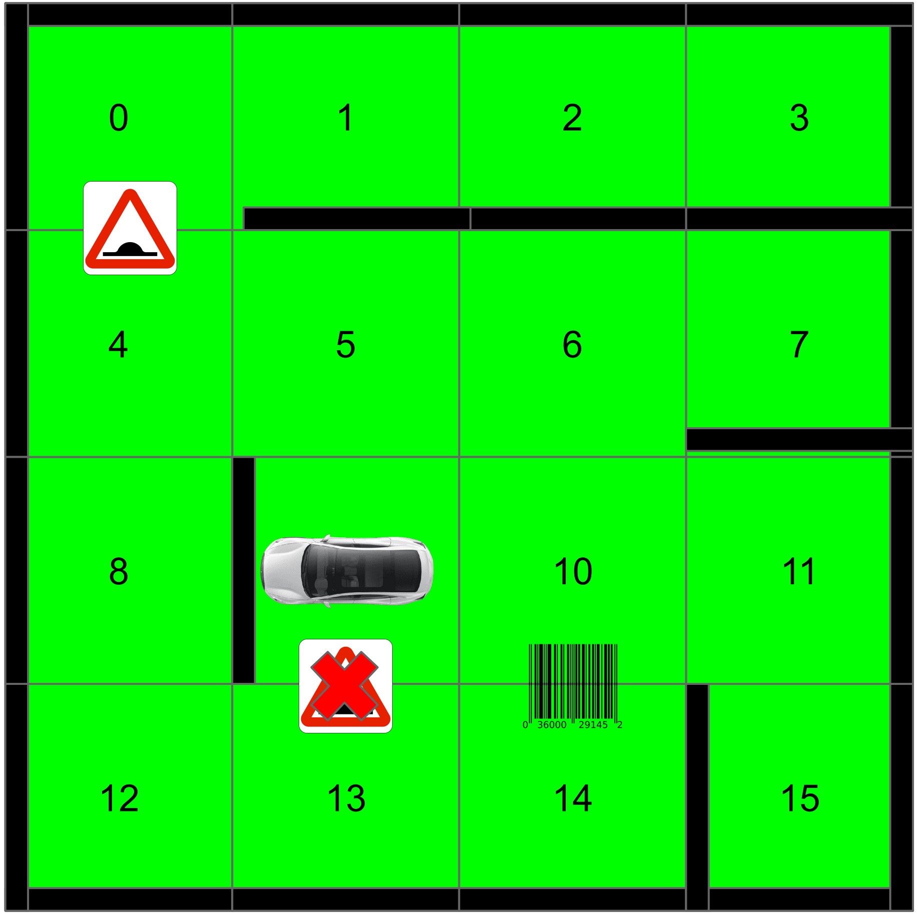


Map Example

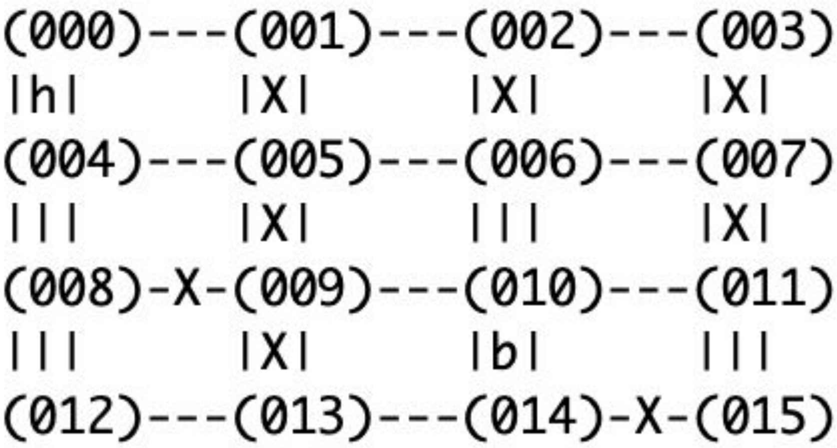
Movement Priority
FORWARD
RIGHT
LEFT
REVERSE (SENSOR THAT YOU CAME FROM IS BLOCKED)

Current Location - 9
Current Orientation - West
Grid Traveled =
[10,6,7,5,4,0,1,2,3,8,12,13,14,11,15,9]

Missed out a hump



Printed Map



```
(000)---(001)---(002)---(003)
 |h|     |X|     |X|     |X|
(004)---(005)---(006)---(007)
 |||     |X|     |||     |X|
(008)-X-(009)---(010)---(011)
 |||     |X|     |b|     |||
(012)---(013)---(014)-X-(015)
```

X - Walls

h - Humps

b - Barcodes

Lines - Accessible

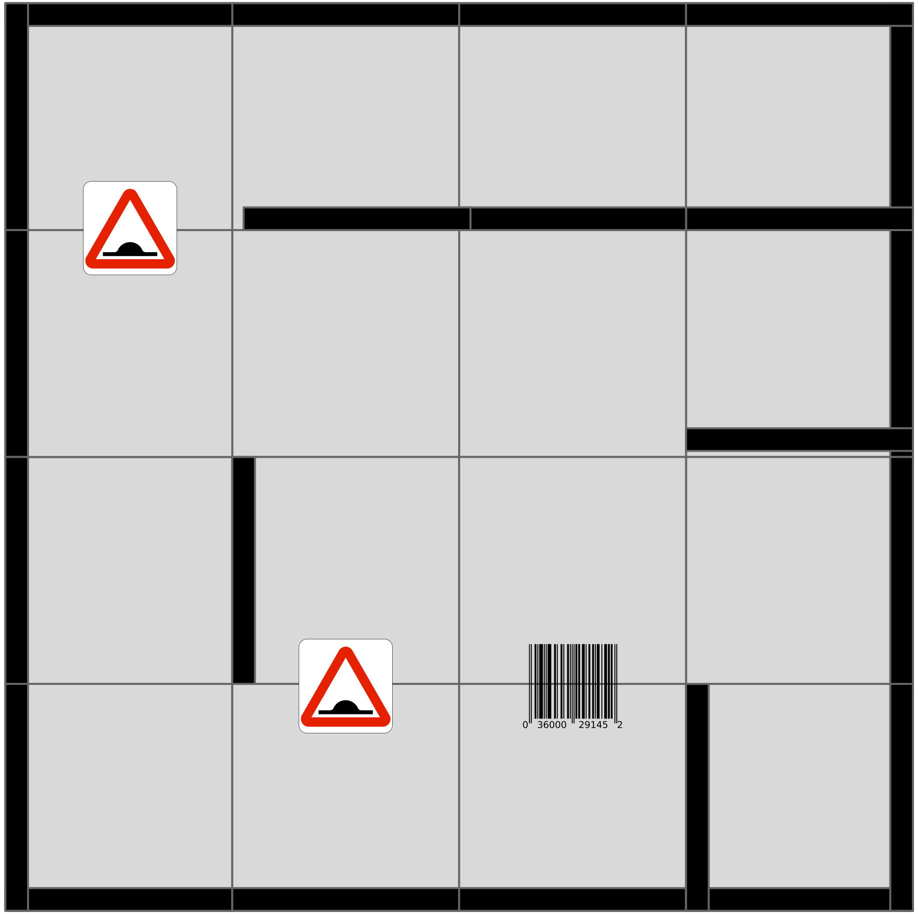
Map Example

Walls

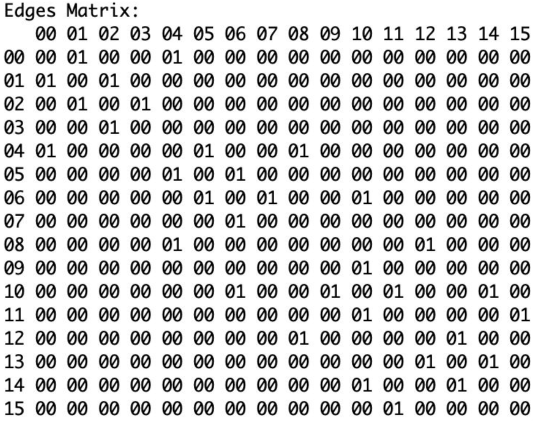
Barcode

Hump

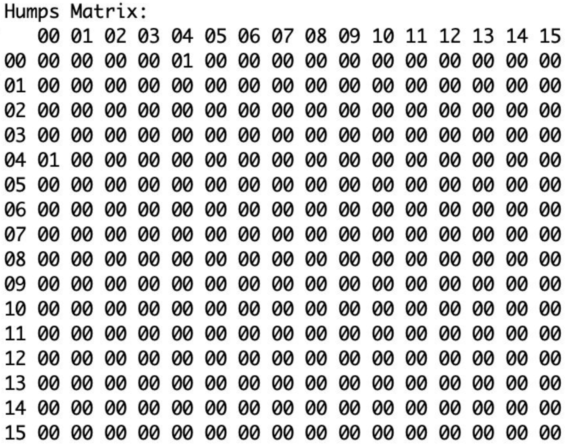
Car



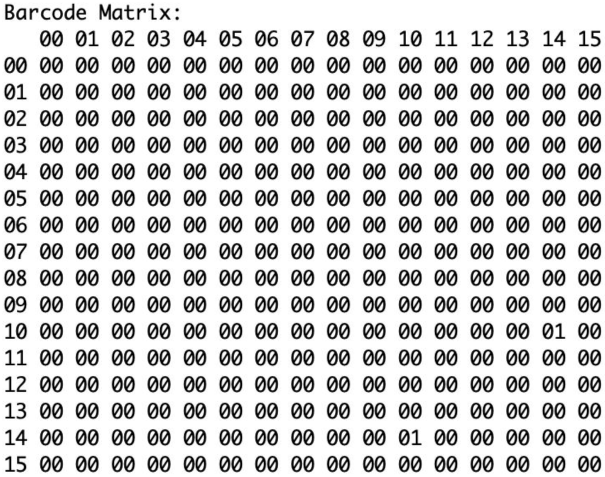
Adjacency Matrix Results



```
Edges Matrix:
   00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 00 01 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
02 00 01 00 01 00 00 00 00 00 00 00 00 00 00 00 00
03 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
04 01 00 00 00 00 01 00 00 01 00 00 00 00 00 00 00
05 00 00 00 00 01 00 01 00 00 00 00 00 00 00 00 00
06 00 00 00 00 00 01 00 01 00 00 01 00 00 00 00 00
07 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00
08 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00
09 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00
10 00 00 00 00 00 00 01 00 00 01 00 01 00 00 01 00
11 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 01
12 00 00 00 00 00 00 00 00 01 00 00 00 00 01 00 00
13 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 00
14 00 00 00 00 00 00 00 00 00 00 01 00 00 01 00 00
15 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00
```

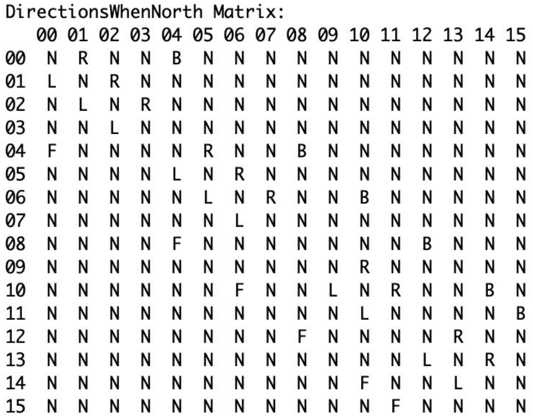


```
Humps Matrix:
   00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
09 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

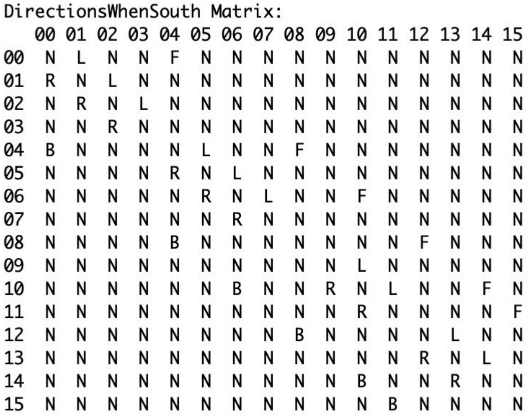


```
Barcode Matrix:
   00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
09 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00
11 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
13 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
14 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00
15 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

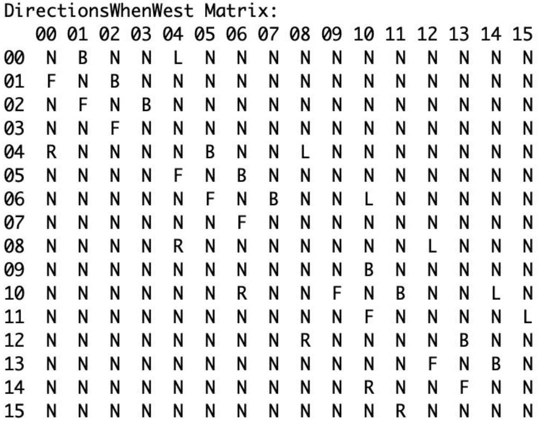
Adjacency Matrix Results



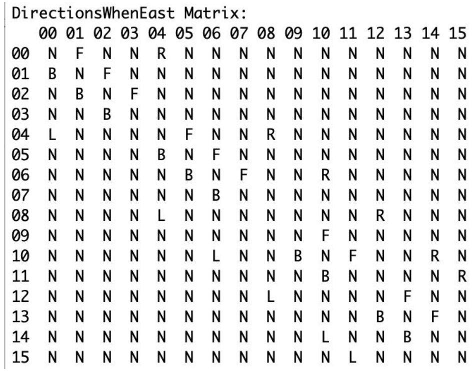
```
DirectionsWhenNorth Matrix:
    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00  N  R  N  N  B  N  N  N  N  N  N  N  N  N  N  N
01  L  N  R  N  N  N  N  N  N  N  N  N  N  N  N  N
02  N  L  N  R  N  N  N  N  N  N  N  N  N  N  N  N
03  N  N  L  N  N  N  N  N  N  N  N  N  N  N  N  N
04  F  N  N  N  N  R  N  N  B  N  N  N  N  N  N  N
05  N  N  N  N  L  N  R  N  N  N  N  N  N  N  N  N
06  N  N  N  N  N  L  N  R  N  N  B  N  N  N  N  N
07  N  N  N  N  N  N  L  N  N  N  N  N  N  N  N  N
08  N  N  N  N  F  N  N  N  N  N  N  N  B  N  N  N
09  N  N  N  N  N  N  N  N  N  N  R  N  N  N  N  N
10  N  N  N  N  N  N  F  N  N  L  N  R  N  N  B  N
11  N  N  N  N  N  N  N  N  N  N  L  N  N  N  N  B
12  N  N  N  N  N  N  N  N  F  N  N  N  N  R  N  N
13  N  N  N  N  N  N  N  N  N  N  N  N  L  N  R  N
14  N  N  N  N  N  N  N  N  N  N  F  N  N  L  N  N
15  N  N  N  N  N  N  N  N  N  N  N  F  N  N  N  N
```



```
DirectionsWhenSouth Matrix:
    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00  N  L  N  N  F  N  N  N  N  N  N  N  N  N  N  N
01  R  N  L  N  N  N  N  N  N  N  N  N  N  N  N  N
02  N  R  N  L  N  N  N  N  N  N  N  N  N  N  N  N
03  N  N  R  N  N  N  N  N  N  N  N  N  N  N  N  N
04  B  N  N  N  N  L  N  N  F  N  N  N  N  N  N  N
05  N  N  N  N  R  N  L  N  N  N  N  N  N  N  N  N
06  N  N  N  N  N  R  N  L  N  N  F  N  N  N  N  N
07  N  N  N  N  N  N  R  N  N  N  N  N  N  N  N  N
08  N  N  N  N  B  N  N  N  N  N  N  N  F  N  N  N
09  N  N  N  N  N  N  N  N  N  N  L  N  N  N  N  N
10  N  N  N  N  N  N  B  N  N  R  N  L  N  N  F  N
11  N  N  N  N  N  N  N  N  N  N  R  N  N  N  N  F
12  N  N  N  N  N  N  N  N  B  N  N  N  N  L  N  N
13  N  N  N  N  N  N  N  N  N  N  N  N  R  N  L  N
14  N  N  N  N  N  N  N  N  N  N  B  N  N  R  N  N
15  N  N  N  N  N  N  N  N  N  N  N  B  N  N  N  N
```

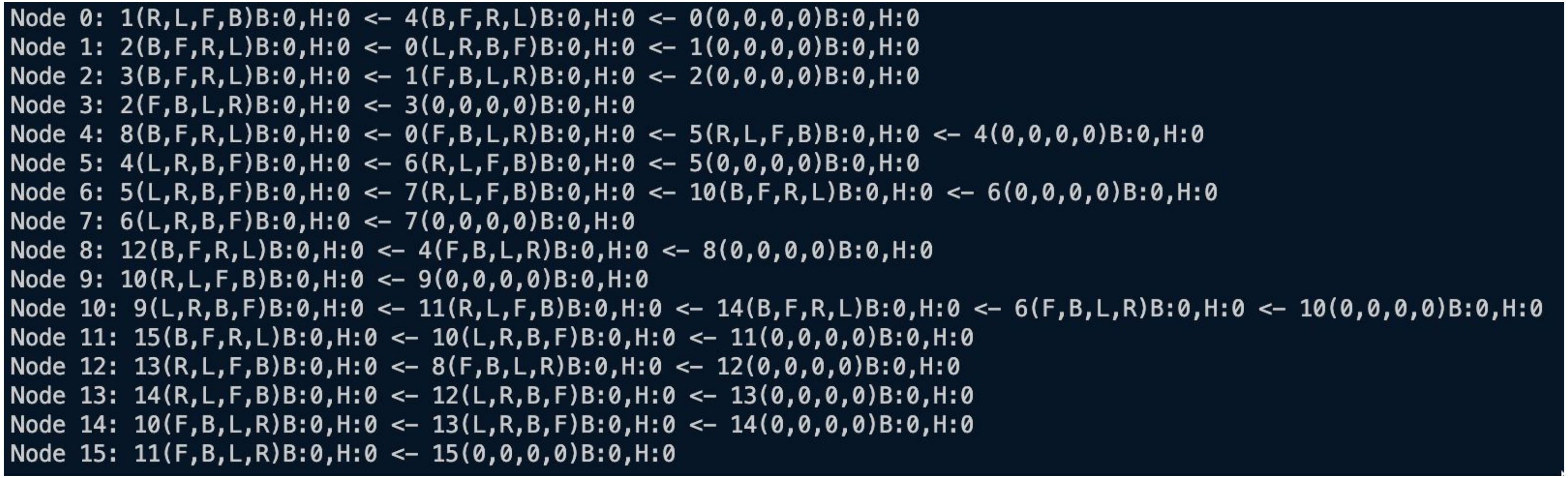


```
DirectionsWhenWest Matrix:
    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00  N  B  N  N  L  N  N  N  N  N  N  N  N  N  N  N
01  F  N  B  N  N  N  N  N  N  N  N  N  N  N  N  N
02  N  F  N  B  N  N  N  N  N  N  N  N  N  N  N  N
03  N  N  F  N  N  N  N  N  N  N  N  N  N  N  N  N
04  R  N  N  N  N  B  N  N  L  N  N  N  N  N  N  N
05  N  N  N  N  F  N  B  N  N  N  N  N  N  N  N  N
06  N  N  N  N  N  F  N  B  N  N  L  N  N  N  N  N
07  N  N  N  N  N  N  F  N  N  N  N  N  N  N  N  N
08  N  N  N  N  R  N  N  N  N  N  N  N  L  N  N  N
09  N  N  N  N  N  N  N  N  N  N  B  N  N  N  N  N
10  N  N  N  N  N  N  R  N  N  F  N  B  N  N  L  N
11  N  N  N  N  N  N  N  N  N  N  F  N  N  N  N  L
12  N  N  N  N  N  N  N  N  R  N  N  N  N  B  N  N
13  N  N  N  N  N  N  N  N  N  N  N  N  F  N  B  N
14  N  N  N  N  N  N  N  N  N  N  R  N  N  F  N  N
15  N  N  N  N  N  N  N  N  N  N  N  R  N  N  N  N
```



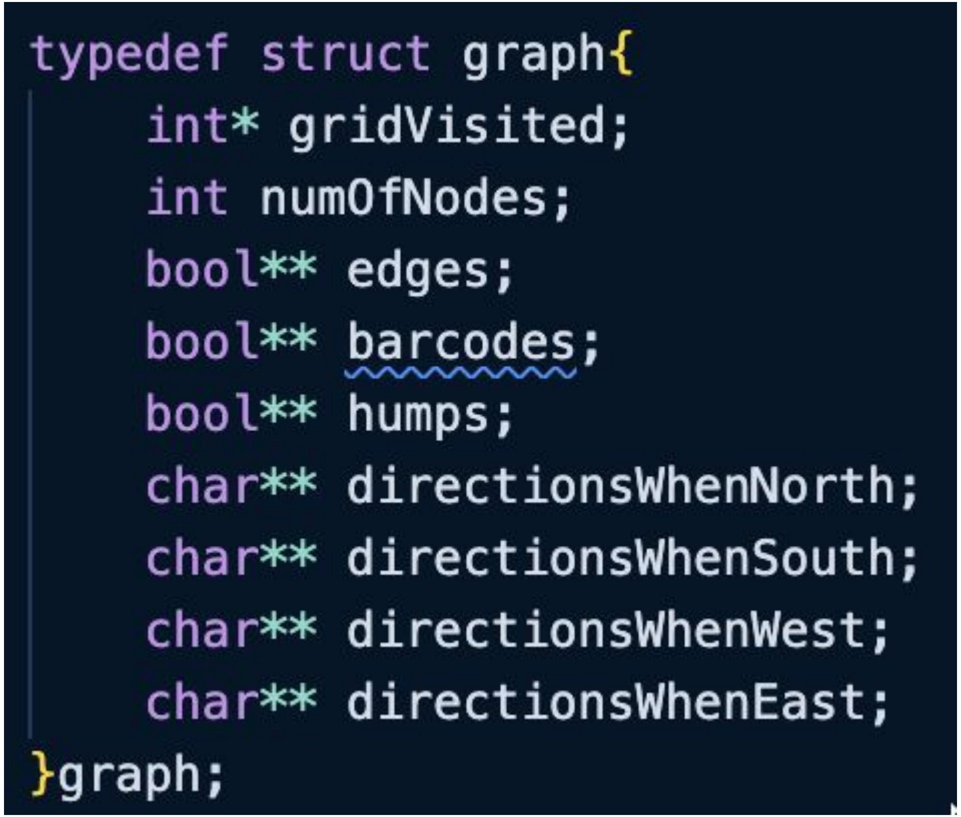
```
DirectionsWhenEast Matrix:
    00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00  N  F  N  N  R  N  N  N  N  N  N  N  N  N  N  N
01  B  N  F  N  N  N  N  N  N  N  N  N  N  N  N  N
02  N  B  N  F  N  N  N  N  N  N  N  N  N  N  N  N
03  N  N  B  N  N  N  N  N  N  N  N  N  N  N  N  N
04  L  N  N  N  N  F  N  N  R  N  N  N  N  N  N  N
05  N  N  N  N  B  N  F  N  N  N  N  N  N  N  N  N
06  N  N  N  N  N  B  N  F  N  N  R  N  N  N  N  N
07  N  N  N  N  N  N  B  N  N  N  N  N  N  N  N  N
08  N  N  N  N  L  N  N  N  N  N  N  N  R  N  N  N
09  N  N  N  N  N  N  N  N  N  N  F  N  N  N  N  N
10  N  N  N  N  N  N  L  N  N  B  N  F  N  N  R  N
11  N  N  N  N  N  N  N  N  N  N  B  N  N  N  N  R
12  N  N  N  N  N  N  N  N  L  N  N  N  N  F  N  N
13  N  N  N  N  N  N  N  N  N  N  N  N  B  N  F  N
14  N  N  N  N  N  N  N  N  N  N  L  N  N  B  N  N
15  N  N  N  N  N  N  N  N  N  N  N  L  N  N  N  N
```

Adjacency List Results



```
Node 0: 1(R,L,F,B)B:0,H:0 <- 4(B,F,R,L)B:0,H:0 <- 0(0,0,0,0)B:0,H:0
Node 1: 2(B,F,R,L)B:0,H:0 <- 0(L,R,B,F)B:0,H:0 <- 1(0,0,0,0)B:0,H:0
Node 2: 3(B,F,R,L)B:0,H:0 <- 1(F,B,L,R)B:0,H:0 <- 2(0,0,0,0)B:0,H:0
Node 3: 2(F,B,L,R)B:0,H:0 <- 3(0,0,0,0)B:0,H:0
Node 4: 8(B,F,R,L)B:0,H:0 <- 0(F,B,L,R)B:0,H:0 <- 5(R,L,F,B)B:0,H:0 <- 4(0,0,0,0)B:0,H:0
Node 5: 4(L,R,B,F)B:0,H:0 <- 6(R,L,F,B)B:0,H:0 <- 5(0,0,0,0)B:0,H:0
Node 6: 5(L,R,B,F)B:0,H:0 <- 7(R,L,F,B)B:0,H:0 <- 10(B,F,R,L)B:0,H:0 <- 6(0,0,0,0)B:0,H:0
Node 7: 6(L,R,B,F)B:0,H:0 <- 7(0,0,0,0)B:0,H:0
Node 8: 12(B,F,R,L)B:0,H:0 <- 4(F,B,L,R)B:0,H:0 <- 8(0,0,0,0)B:0,H:0
Node 9: 10(R,L,F,B)B:0,H:0 <- 9(0,0,0,0)B:0,H:0
Node 10: 9(L,R,B,F)B:0,H:0 <- 11(R,L,F,B)B:0,H:0 <- 14(B,F,R,L)B:0,H:0 <- 6(F,B,L,R)B:0,H:0 <- 10(0,0,0,0)B:0,H:0
Node 11: 15(B,F,R,L)B:0,H:0 <- 10(L,R,B,F)B:0,H:0 <- 11(0,0,0,0)B:0,H:0
Node 12: 13(R,L,F,B)B:0,H:0 <- 8(F,B,L,R)B:0,H:0 <- 12(0,0,0,0)B:0,H:0
Node 13: 14(R,L,F,B)B:0,H:0 <- 12(L,R,B,F)B:0,H:0 <- 13(0,0,0,0)B:0,H:0
Node 14: 10(F,B,L,R)B:0,H:0 <- 13(L,R,B,F)B:0,H:0 <- 14(0,0,0,0)B:0,H:0
Node 15: 11(F,B,L,R)B:0,H:0 <- 15(0,0,0,0)B:0,H:0
```

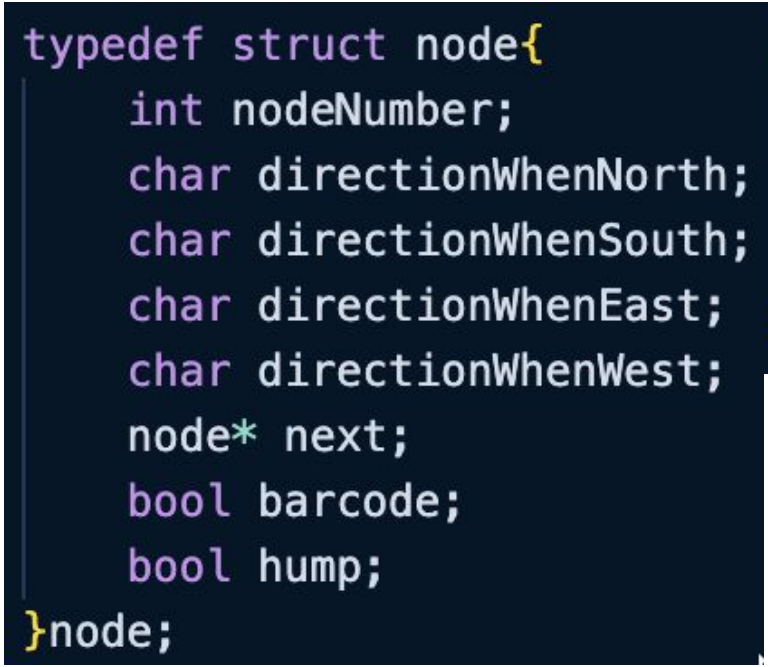
Adjacency Matrix (4 X 4 Example)



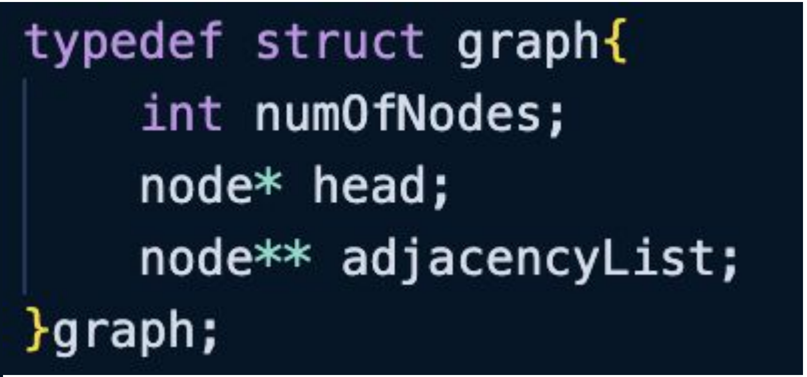
```
typedef struct graph{
        int* gridVisited;
        int numOfNodes;
        bool** edges;
        bool** barcodes;
        bool** humps;
        char** directionsWhenNorth;
        char** directionsWhenSouth;
        char** directionsWhenWest;
        char** directionsWhenEast;
}graph;
```

8 bytes	4 bytes X 16 grids = 64
4 bytes	
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
8 bytes	1 bytes X 256 elements = 256
	1860 bytes

Adjacency List (4 X 4 Example)



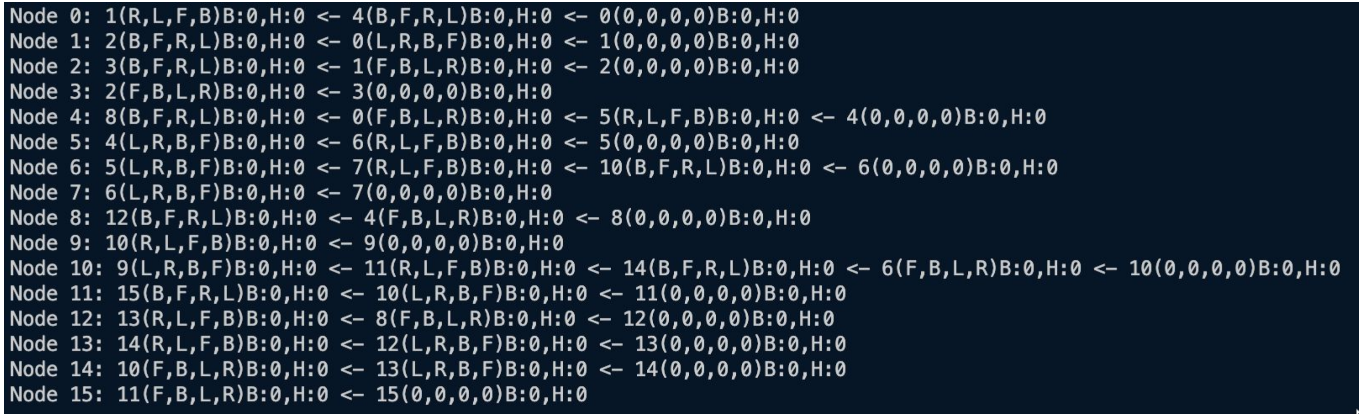
```
typedef struct node{
    int nodeNumber;
    char directionWhenNorth;
    char directionWhenSouth;
    char directionWhenEast;
    char directionWhenWest;
    node* next;
    bool barcode;
    bool hump;
}node;
```



```
typedef struct graph{
    int numOfNodes;
    node* head;
    node** adjacencyList;
}graph;
```

Each Node = 18 bytes
Graph Struct = 20 bytes

Adjacency List (4 X 4 Example)

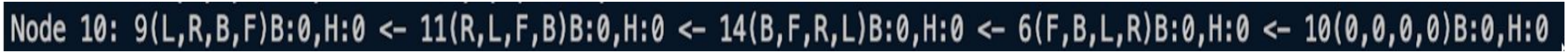


```
Node 0: 1(R,L,F,B)B:0,H:0 <- 4(B,F,R,L)B:0,H:0 <- 0(0,0,0,0)B:0,H:0
Node 1: 2(B,F,R,L)B:0,H:0 <- 0(L,R,B,F)B:0,H:0 <- 1(0,0,0,0)B:0,H:0
Node 2: 3(B,F,R,L)B:0,H:0 <- 1(F,B,L,R)B:0,H:0 <- 2(0,0,0,0)B:0,H:0
Node 3: 2(F,B,L,R)B:0,H:0 <- 3(0,0,0,0)B:0,H:0
Node 4: 8(B,F,R,L)B:0,H:0 <- 0(F,B,L,R)B:0,H:0 <- 5(R,L,F,B)B:0,H:0 <- 4(0,0,0,0)B:0,H:0
Node 5: 4(L,R,B,F)B:0,H:0 <- 6(R,L,F,B)B:0,H:0 <- 5(0,0,0,0)B:0,H:0
Node 6: 5(L,R,B,F)B:0,H:0 <- 7(R,L,F,B)B:0,H:0 <- 10(B,F,R,L)B:0,H:0 <- 6(0,0,0,0)B:0,H:0
Node 7: 6(L,R,B,F)B:0,H:0 <- 7(0,0,0,0)B:0,H:0
Node 8: 12(B,F,R,L)B:0,H:0 <- 4(F,B,L,R)B:0,H:0 <- 8(0,0,0,0)B:0,H:0
Node 9: 10(R,L,F,B)B:0,H:0 <- 9(0,0,0,0)B:0,H:0
Node 10: 9(L,R,B,F)B:0,H:0 <- 11(R,L,F,B)B:0,H:0 <- 14(B,F,R,L)B:0,H:0 <- 6(F,B,L,R)B:0,H:0 <- 10(0,0,0,0)B:0,H:0
Node 11: 15(B,F,R,L)B:0,H:0 <- 10(L,R,B,F)B:0,H:0 <- 11(0,0,0,0)B:0,H:0
Node 12: 13(R,L,F,B)B:0,H:0 <- 8(F,B,L,R)B:0,H:0 <- 12(0,0,0,0)B:0,H:0
Node 13: 14(R,L,F,B)B:0,H:0 <- 12(L,R,B,F)B:0,H:0 <- 13(0,0,0,0)B:0,H:0
Node 14: 10(F,B,L,R)B:0,H:0 <- 13(L,R,B,F)B:0,H:0 <- 14(0,0,0,0)B:0,H:0
Node 15: 11(F,B,L,R)B:0,H:0 <- 15(0,0,0,0)B:0,H:0
```

882 bytes + 20 bytes(Graph Struct) = 902 bytes

Adjacency List
(0) 18 bytes X 3 = 54
(1) 18 bytes X 3 = 54
(2) 18 bytes X 3 = 54
(3) 18 bytes X 2 =36
(4) 18 bytes X 4 = 72
(5) 18 bytes X 3 = 54
(6) 18 bytes X 4 = 72
(7) 18 bytes X 2 = 36
(8) 18 bytes X 3 = 72
(9) 18 bytes X 2 = 36
(10) 18 bytes X 5 = 90
(11) 18 bytes X 3 = 54
(12) 18 bytes X 3 = 54
(13) 18 bytes X 3 = 54
(14) 18 bytes X 3 = 54
(15) 18 bytes X 2 = 36
Total = 882 bytes

Adjacency Matrix VS Adjacency List



```
Node 10: 9(L,R,B,F)B:0,H:0 <- 11(R,L,F,B)B:0,H:0 <- 14(B,F,R,L)B:0,H:0 <- 6(F,B,L,R)B:0,H:0 <- 10(0,0,0,0)B:0,H:0
```

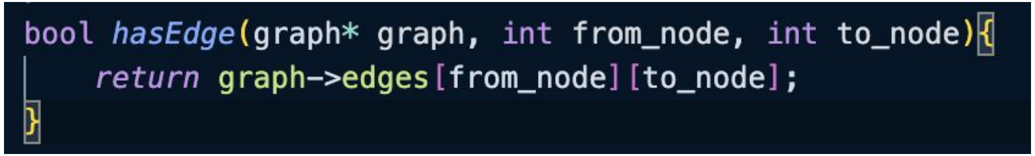
Memory Complexity

Adjacency List

- Consumes less memory (902 bytes)

Adjacency Matrix

- Consumes more memory (1860 bytes)



```
bool hasEdge(graph* graph, int from_node, int to_node){
    return graph->edges[from_node][to_node];
}
```

Lists is more dynamic and efficient when inserting/deleting

When checking if the car can go from grid 10 to 6, it would have to check the whole linked lists that is connected to grid 10

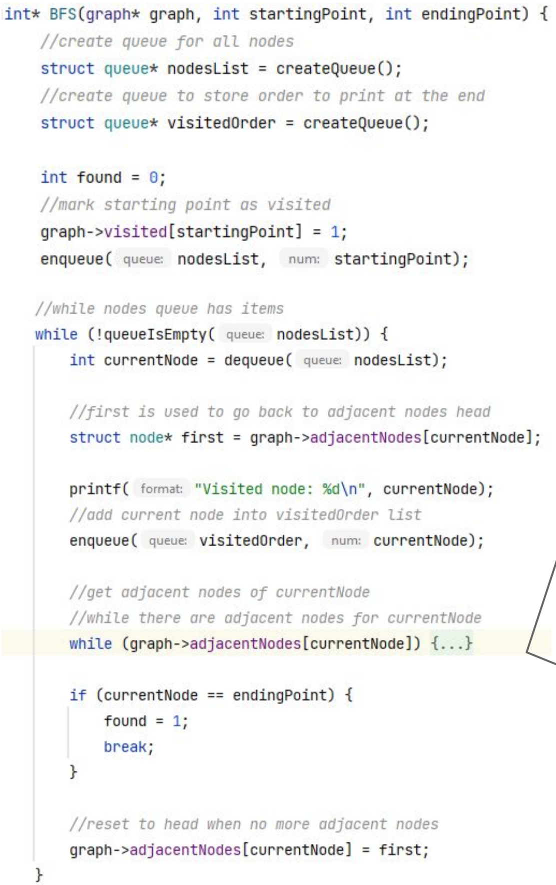
Matrix allows random access, lesser memory per element however can be inefficient when inserting/deleting

When checking if the car can go from grid 10 to 6, we can straight away use hasEdge(map,10,6) to immediately get a result.

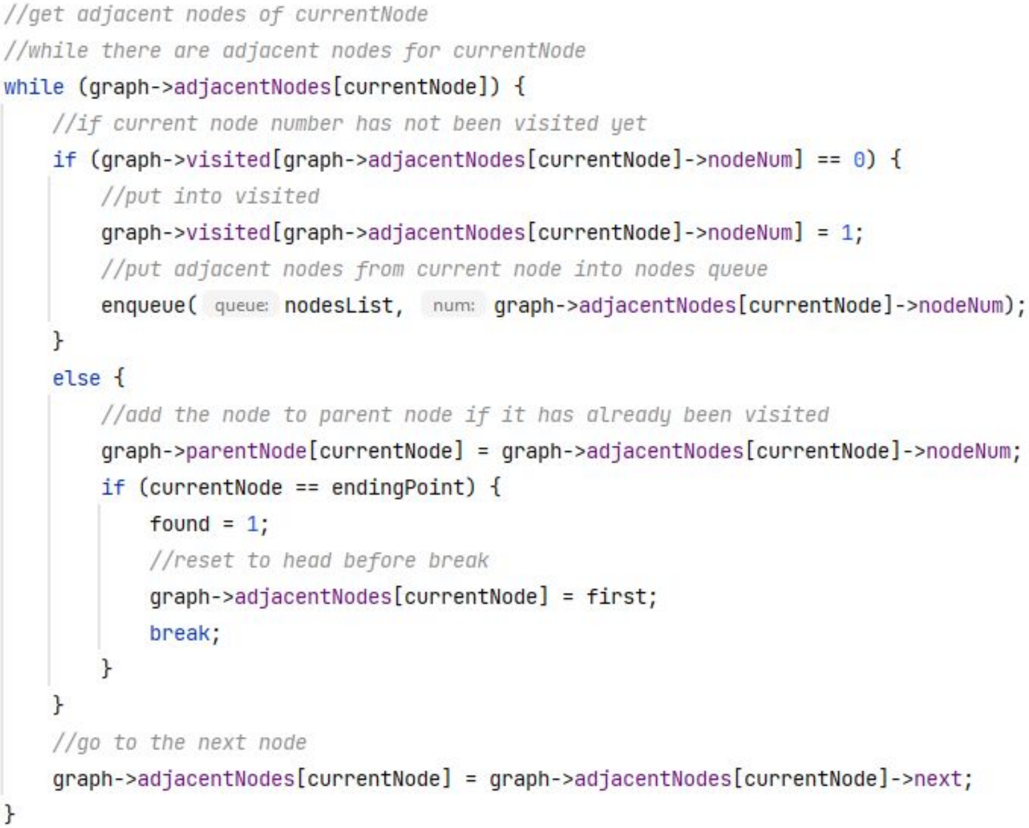
NAVIGATION

BFS Implementation

BFS

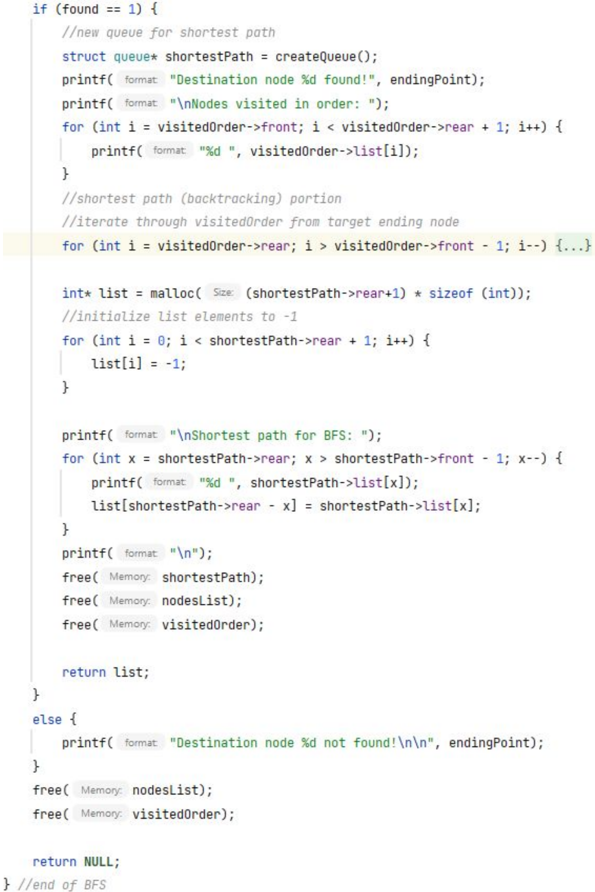


```c
int* BFS(graph* graph, int startingPoint, int endingPoint) {
    //create queue for all nodes
    struct queue* nodesList = createQueue();
    //create queue to store order to print at the end
    struct queue* visitedOrder = createQueue();

    int found = 0;
    //mark starting point as visited
    graph->visited[startingPoint] = 1;
    enqueue( queue: nodesList,  num: startingPoint);

    //while nodes queue has items
    while (!queueIsEmpty( queue: nodesList)) {
        int currentNode = dequeue( queue: nodesList);

        //first is used to go back to adjacent nodes head
        struct node* first = graph->adjacentNodes[currentNode];

        printf( format: "Visited node: %d\n", currentNode);
        //add current node into visitedOrder list
        enqueue( queue: visitedOrder,  num: currentNode);

        //get adjacent nodes of currentNode
        //while there are adjacent nodes for currentNode
        while (graph->adjacentNodes[currentNode]) {...}

        if (currentNode == endingPoint) {
            found = 1;
            break;
        }

        //reset to head when no more adjacent nodes
        graph->adjacentNodes[currentNode] = first;
    }
```

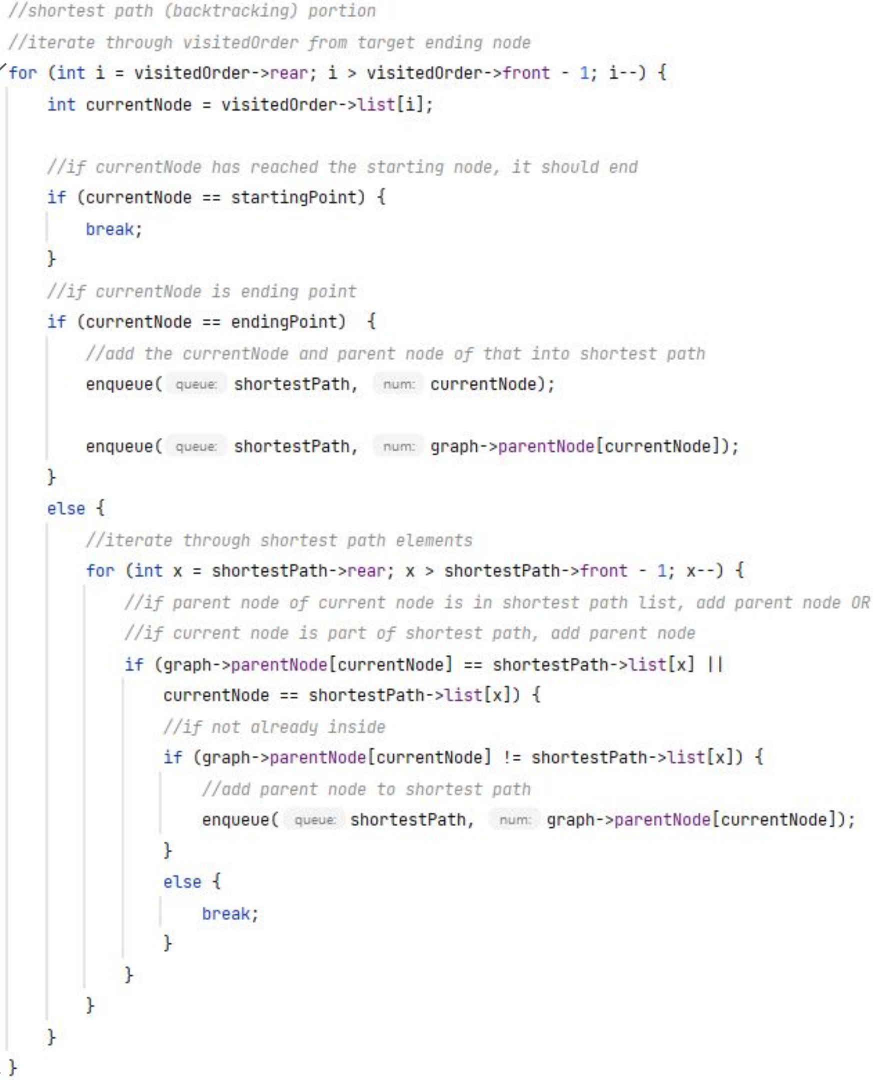


```c
//get adjacent nodes of currentNode
//while there are adjacent nodes for currentNode
while (graph->adjacentNodes[currentNode]) {
    //if current node number has not been visited yet
    if (graph->visited[graph->adjacentNodes[currentNode]->nodeNum] == 0) {
        //put into visited
        graph->visited[graph->adjacentNodes[currentNode]->nodeNum] = 1;
        //put adjacent nodes from current node into nodes queue
        enqueue( queue: nodesList,  num: graph->adjacentNodes[currentNode]->nodeNum);
    }
    else {
        //add the node to parent node if it has already been visited
        graph->parentNode[currentNode] = graph->adjacentNodes[currentNode]->nodeNum;
        if (currentNode == endingPoint) {
            found = 1;
            //reset to head before break
            graph->adjacentNodes[currentNode] = first;
            break;
        }
    }
    //go to the next node
    graph->adjacentNodes[currentNode] = graph->adjacentNodes[currentNode]->next;
}
```

BFS (cont.)



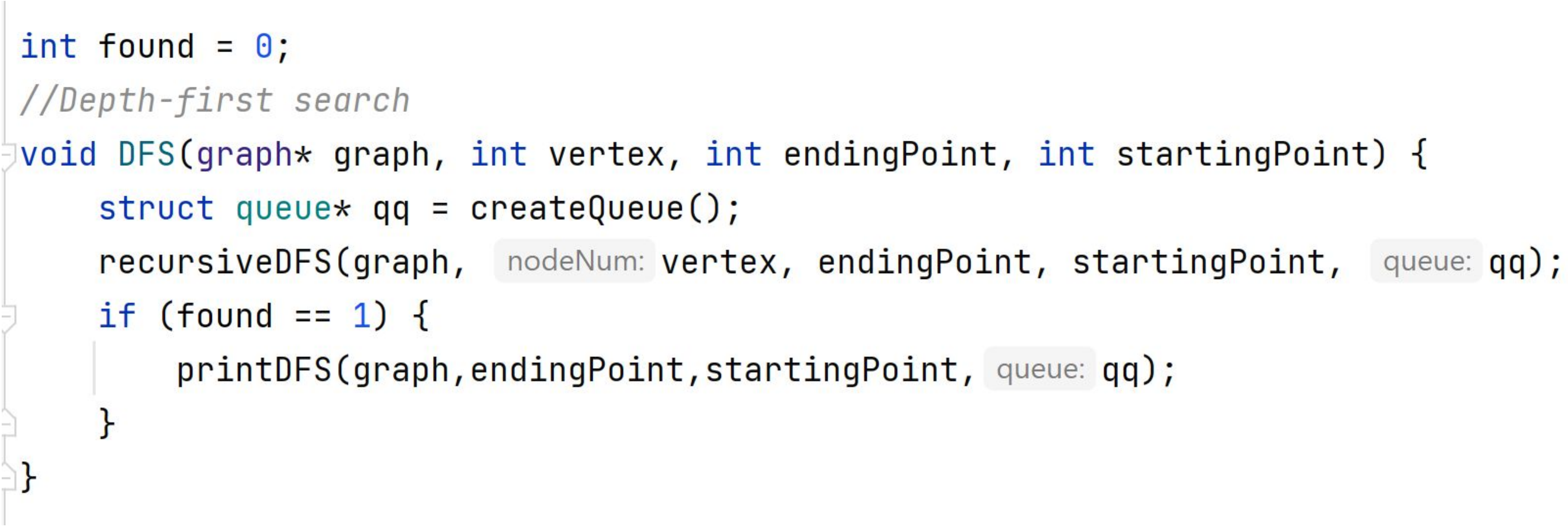
```
if (found == 1) {
    //new queue for shortest path
    struct queue* shortestPath = createQueue();
    printf( format: "Destination node %d found!", endingPoint);
    printf( format: "\nNodes visited in order: ");
    for (int i = visitedOrder->front; i < visitedOrder->rear + 1; i++) {
        printf( format: "%d ", visitedOrder->list[i]);
    }
    //shortest path (backtracking) portion
    //iterate through visitedOrder from target ending node
    for (int i = visitedOrder->rear; i > visitedOrder->front - 1; i--) {...}

    int* list = malloc( Size: (shortestPath->rear+1) * sizeof (int));
    //initialize list elements to -1
    for (int i = 0; i < shortestPath->rear + 1; i++) {
        list[i] = -1;
    }

    printf( format: "\nShortest path for BFS: ");
    for (int x = shortestPath->rear; x > shortestPath->front - 1; x--) {
        printf( format: "%d ", shortestPath->list[x]);
        list[shortestPath->rear - x] = shortestPath->list[x];
    }
    printf( format: "\n");
    free( Memory: shortestPath);
    free( Memory: nodesList);
    free( Memory: visitedOrder);

    return list;
}
else {
    printf( format: "Destination node %d not found!\n\n", endingPoint);
}
free( Memory: nodesList);
free( Memory: visitedOrder);

return NULL;
} //end of BFS
```



```
//shortest path (backtracking) portion
//iterate through visitedOrder from target ending node
for (int i = visitedOrder->rear; i > visitedOrder->front - 1; i--) {
    int currentNode = visitedOrder->list[i];

    //if currentNode has reached the starting node, it should end
    if (currentNode == startingPoint) {
        break;
    }
    //if currentNode is ending point
    if (currentNode == endingPoint)  {
        //add the currentNode and parent node of that into shortest path
        enqueue( queue: shortestPath,  num: currentNode);

        enqueue( queue: shortestPath,  num: graph->parentNode[currentNode]);
    }
    else {
        //iterate through shortest path elements
        for (int x = shortestPath->rear; x > shortestPath->front - 1; x--) {
            //if parent node of current node is in shortest path list, add parent node OR
            //if current node is part of shortest path, add parent node
            if (graph->parentNode[currentNode] == shortestPath->list[x] ||
                currentNode == shortestPath->list[x]) {
                //if not already inside
                if (graph->parentNode[currentNode] != shortestPath->list[x]) {
                    //add parent node to shortest path
                    enqueue( queue: shortestPath,  num: graph->parentNode[currentNode]);
                }
                else {
                    break;
                }
            }
        }
    }
}
```

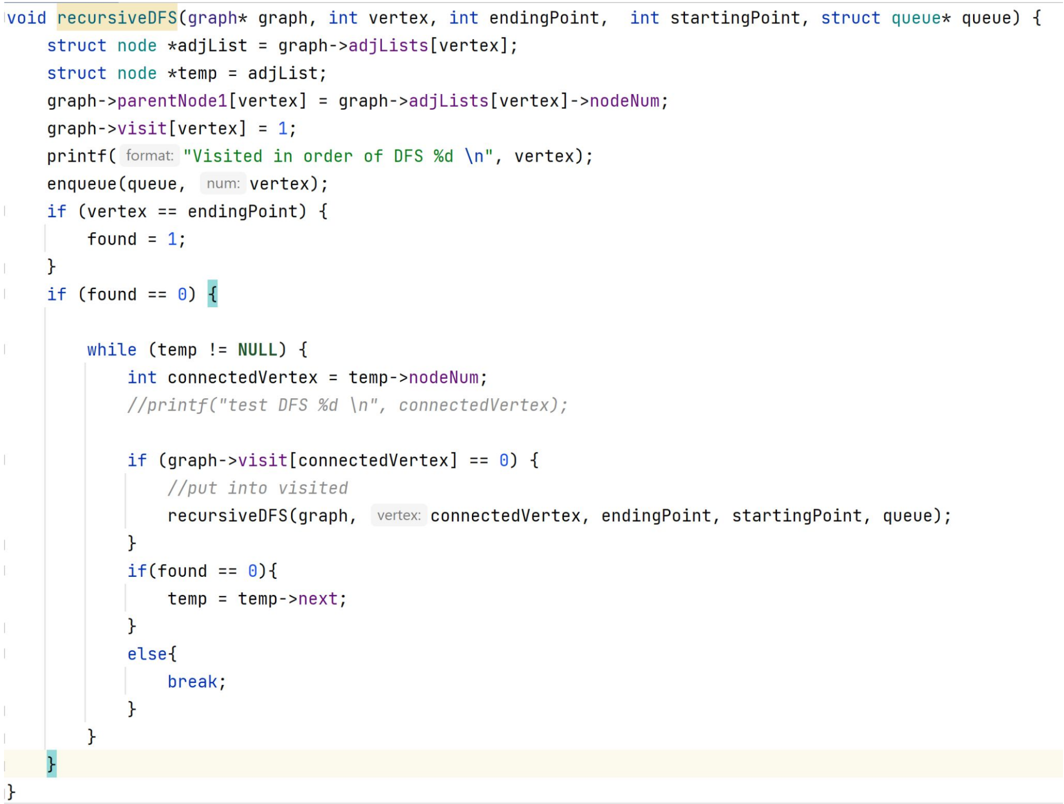
DFS Implementation

DFS



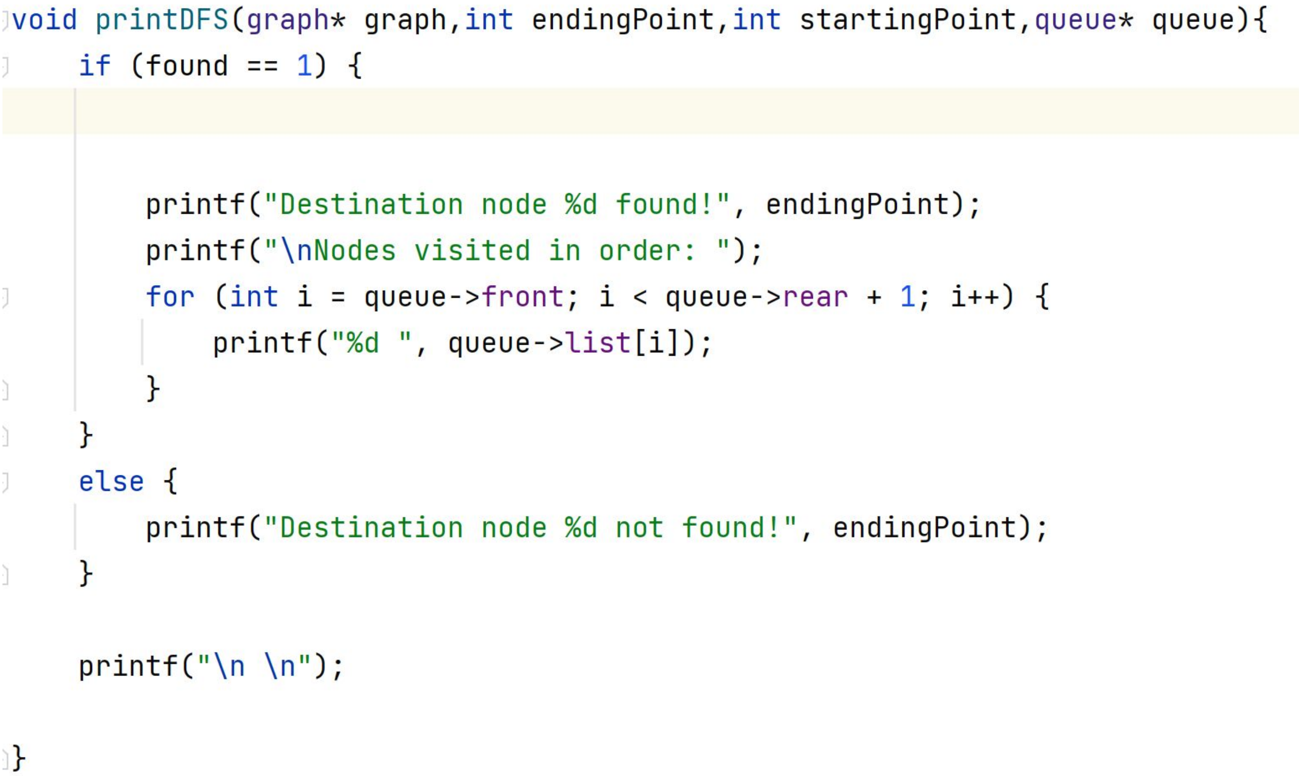
```
int found = 0;
//Depth-first search
void DFS(graph* graph, int vertex, int endingPoint, int startingPoint) {
    struct queue* qq = createQueue();
    recursiveDFS(graph, nodeNum: vertex, endingPoint, startingPoint, queue: qq);
    if (found == 1) {
        printDFS(graph,endingPoint,startingPoint, queue: qq);
    }
}
```

DFS (cont.)



```
void recursiveDFS(graph* graph, int vertex, int endingPoint,  int startingPoint, struct queue* queue) {
    struct node *adjList = graph->adjLists[vertex];
    struct node *temp = adjList;
    graph->parentNode1[vertex] = graph->adjLists[vertex]->nodeNum;
    graph->visit[vertex] = 1;
    printf( format: "Visited in order of DFS %d \n", vertex);
    enqueue(queue,  num: vertex);
    if (vertex == endingPoint) {
        found = 1;
    }
    if (found == 0) {

        while (temp != NULL) {
            int connectedVertex = temp->nodeNum;
            //printf("test DFS %d \n", connectedVertex);

            if (graph->visit[connectedVertex] == 0) {
                //put into visited
                recursiveDFS(graph,  vertex: connectedVertex, endingPoint, startingPoint, queue);
            }
            if(found == 0){
                temp = temp->next;
            }
            else{
                break;
            }
        }
    }
}
```

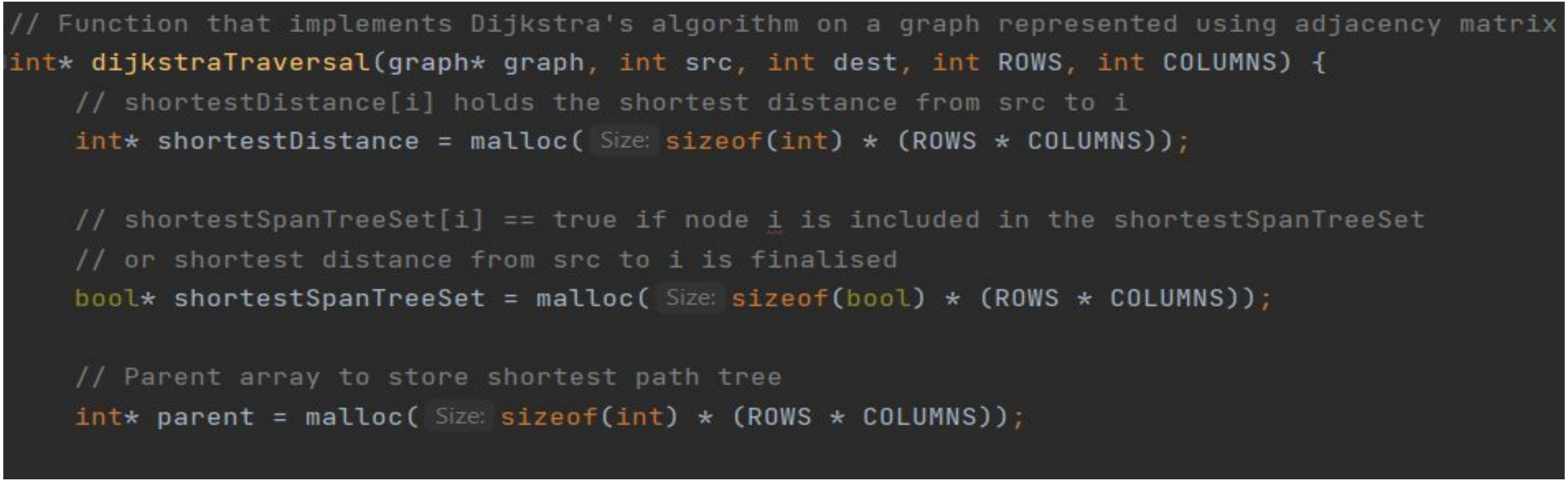
DFS



```
void printDFS(graph* graph,int endingPoint,int startingPoint,queue* queue){
    if (found == 1) {


        printf("Destination node %d found!", endingPoint);
        printf("\nNodes visited in order: ");
        for (int i = queue->front; i < queue->rear + 1; i++) {
            printf("%d ", queue->list[i]);
        }
    }
    else {
        printf("Destination node %d not found!", endingPoint);
    }

    printf("\n \n");

}
```

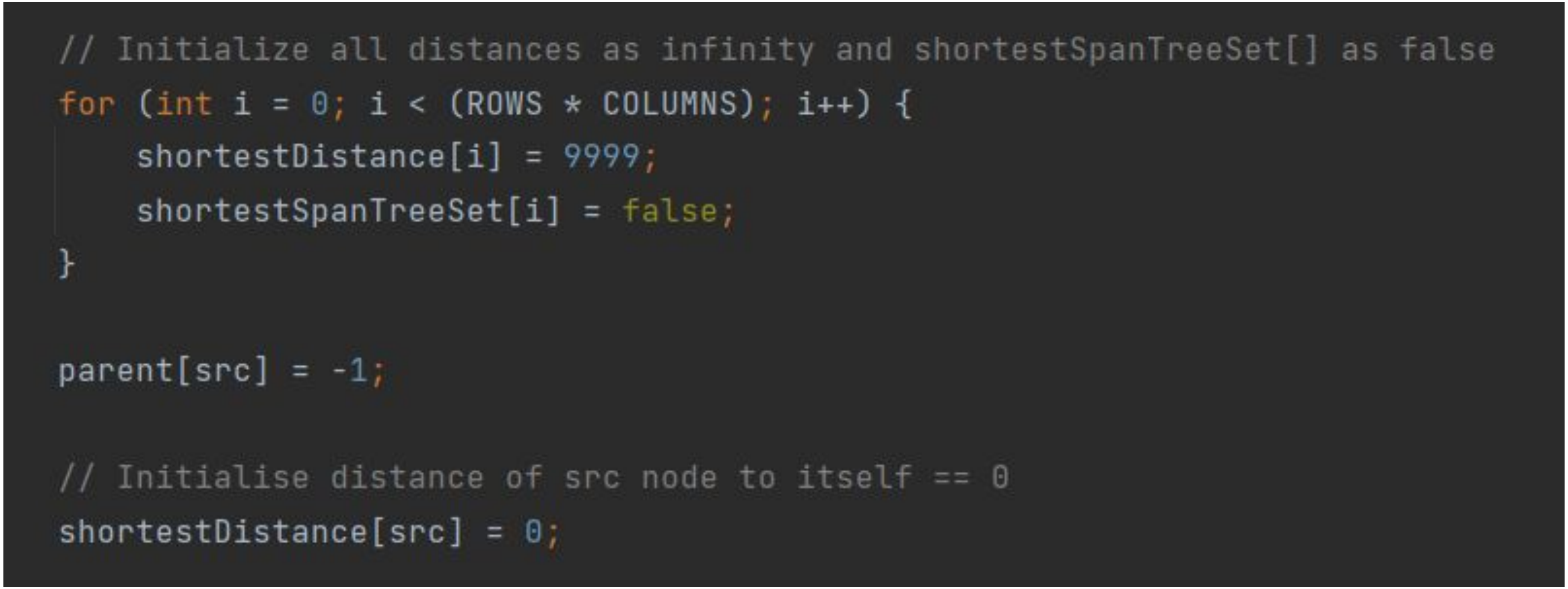
Dijkstra Implementation

Dijkstra



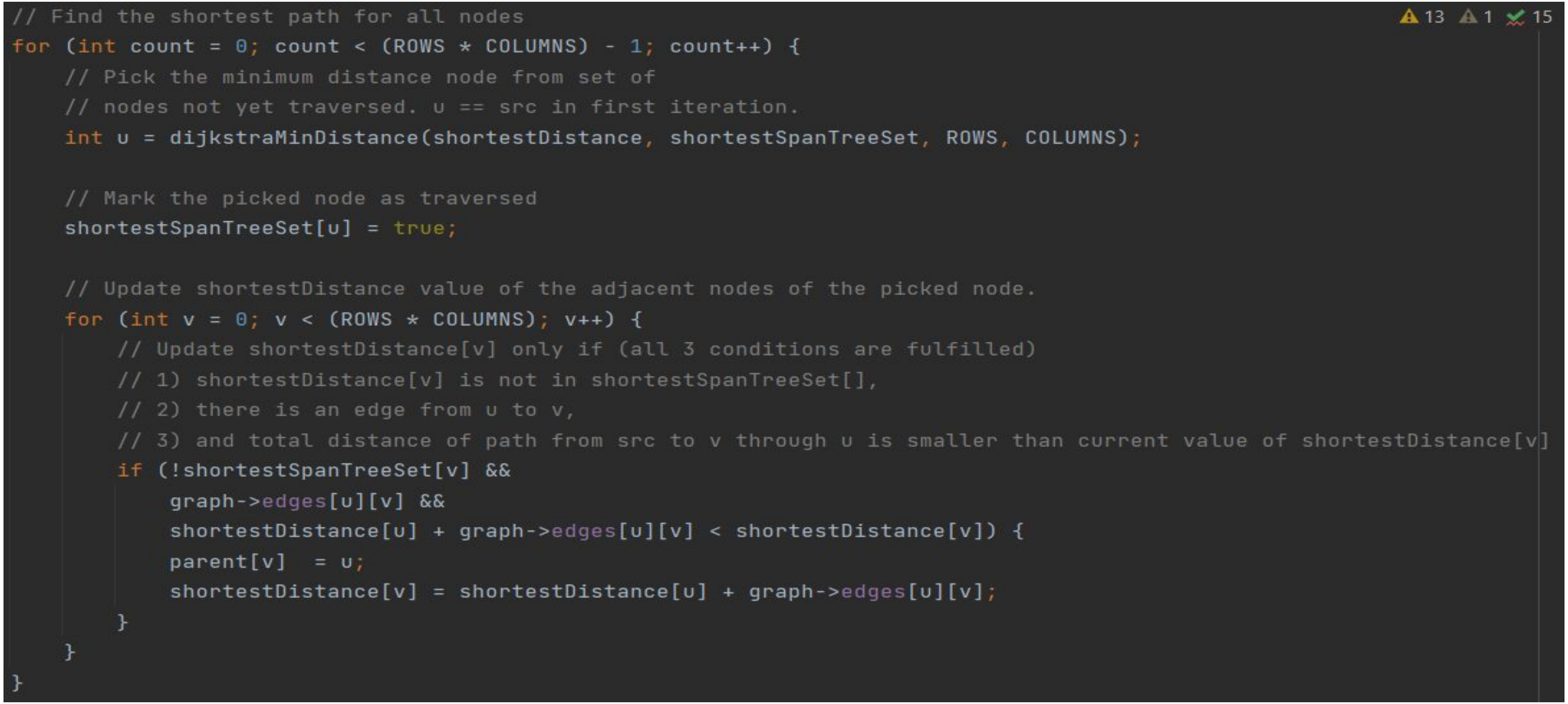
```
// Function that implements Dijkstra's algorithm on a graph represented using adjacency matrix
int* dijkstraTraversal(graph* graph, int src, int dest, int ROWS, int COLUMNS) {
    // shortestDistance[i] holds the shortest distance from src to i
    int* shortestDistance = malloc( Size: sizeof(int) * (ROWS * COLUMNS));

    // shortestSpanTreeSet[i] == true if node i is included in the shortestSpanTreeSet
    // or shortest distance from src to i is finalised
    bool* shortestSpanTreeSet = malloc( Size: sizeof(bool) * (ROWS * COLUMNS));

    // Parent array to store shortest path tree
    int* parent = malloc( Size: sizeof(int) * (ROWS * COLUMNS));
```

Dijkstra (cont.)



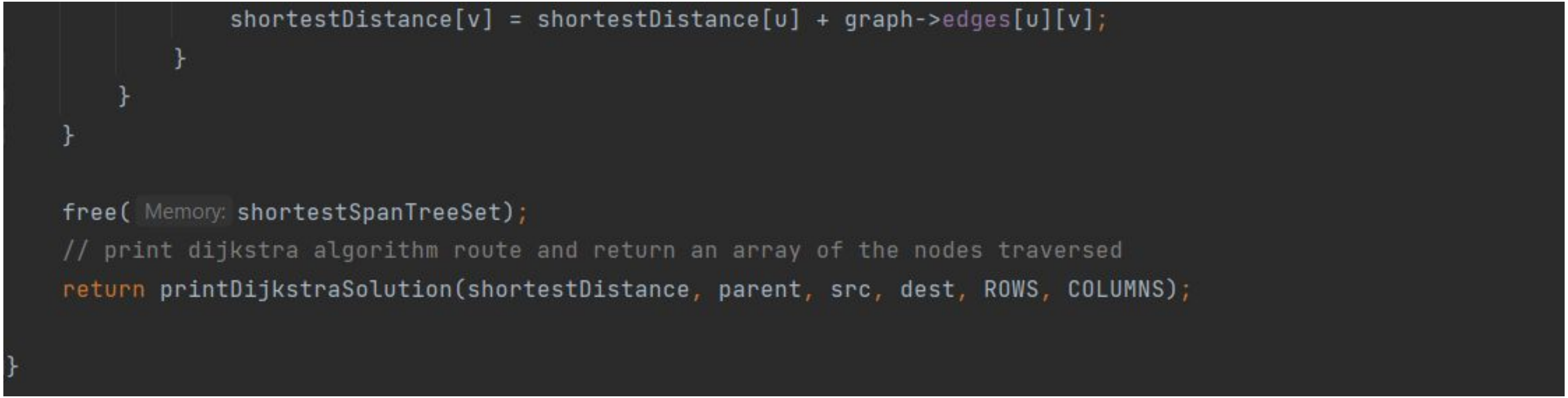
```
// Initialize all distances as infinity and shortestSpanTreeSet[] as false
for (int i = 0; i < (ROWS * COLUMNS); i++) {
    shortestDistance[i] = 9999;
    shortestSpanTreeSet[i] = false;
}

parent[src] = -1;

// Initialise distance of src node to itself == 0
shortestDistance[src] = 0;
```

Dijkstra (cont.)



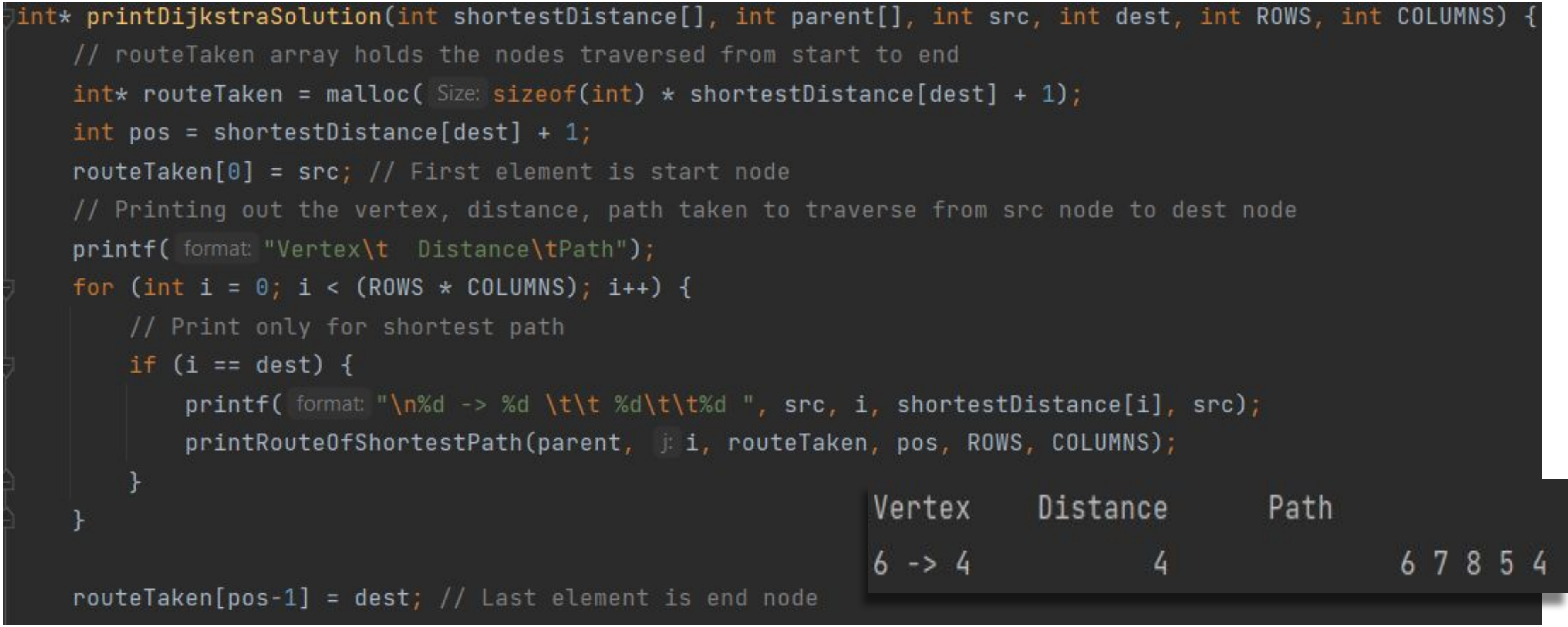
```
// Find the shortest path for all nodes
for (int count = 0; count < (ROWS * COLUMNS) - 1; count++) {
    // Pick the minimum distance node from set of
    // nodes not yet traversed. u == src in first iteration.
    int u = dijkstraMinDistance(shortestDistance, shortestSpanTreeSet, ROWS, COLUMNS);

    // Mark the picked node as traversed
    shortestSpanTreeSet[u] = true;

    // Update shortestDistance value of the adjacent nodes of the picked node.
    for (int v = 0; v < (ROWS * COLUMNS); v++) {
        // Update shortestDistance[v] only if (all 3 conditions are fulfilled)
        // 1) shortestDistance[v] is not in shortestSpanTreeSet[],
        // 2) there is an edge from u to v,
        // 3) and total distance of path from src to v through u is smaller than current value of shortestDistance[v]
        if (!shortestSpanTreeSet[v] &&
            graph->edges[u][v] &&
            shortestDistance[u] + graph->edges[u][v] < shortestDistance[v]) {
            parent[v]  = u;
            shortestDistance[v] = shortestDistance[u] + graph->edges[u][v];
        }
    }
}
```

Dijkstra (cont.)



```
                shortestDistance[v] = shortestDistance[u] + graph->edges[u][v];
            }
        }
    }

    free( Memory: shortestSpanTreeSet);
    // print dijkstra algorithm route and return an array of the nodes traversed
    return printDijkstraSolution(shortestDistance, parent, src, dest, ROWS, COLUMNS);

}
```

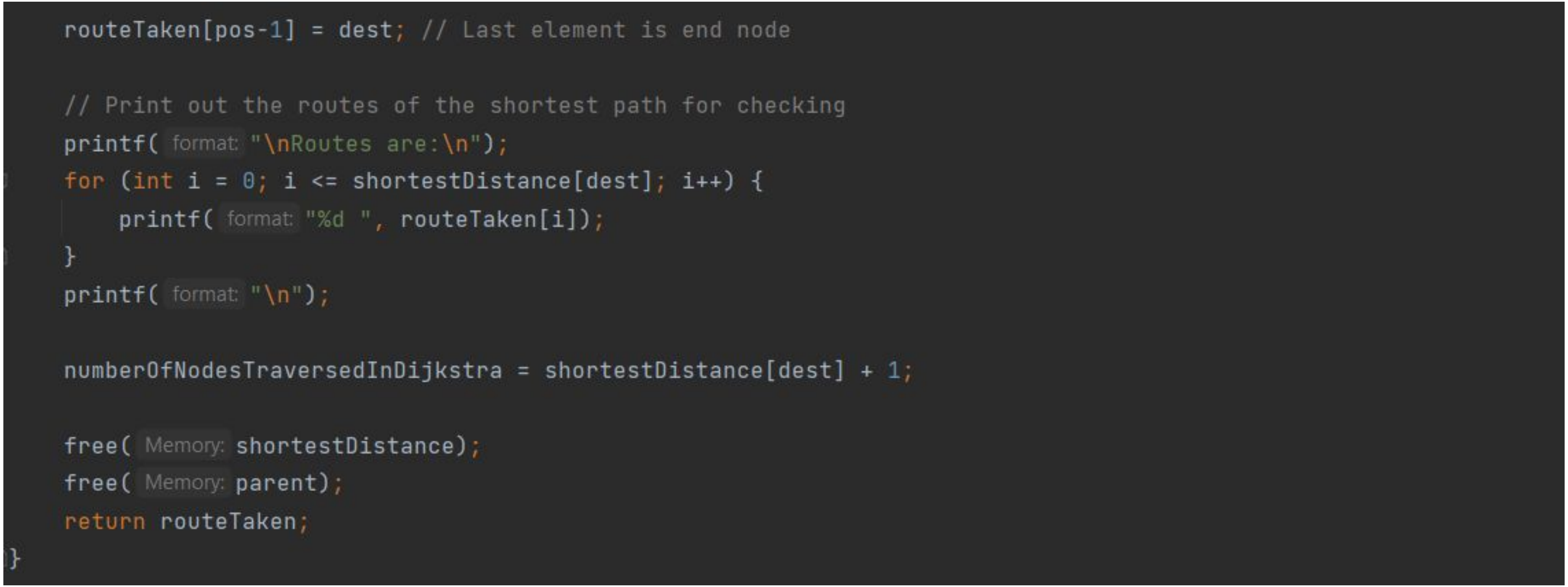
Dijkstra (cont.)



```
int* printDijkstraSolution(int shortestDistance[], int parent[], int src, int dest, int ROWS, int COLUMNS) {
    // routeTaken array holds the nodes traversed from start to end
    int* routeTaken = malloc( Size: sizeof(int) * shortestDistance[dest] + 1);
    int pos = shortestDistance[dest] + 1;
    routeTaken[0] = src; // First element is start node
    // Printing out the vertex, distance, path taken to traverse from src node to dest node
    printf( format: "Vertex\t  Distance\tPath");
    for (int i = 0; i < (ROWS * COLUMNS); i++) {
        // Print only for shortest path
        if (i == dest) {
            printf( format: "\n%d -> %d \t\t %d\t\t%d ", src, i, shortestDistance[i], src);
            printRouteOfShortestPath(parent,  j: i, routeTaken, pos, ROWS, COLUMNS);
        }
    }

    routeTaken[pos-1] = dest; // Last element is end node
```

```
Vertex    Distance      Path
6 -> 4           4              6 7 8 5 4
```

Dijkstra (cont.)



```
    routeTaken[pos-1] = dest; // Last element is end node

    // Print out the routes of the shortest path for checking
    printf( format: "\nRoutes are:\n");
    for (int i = 0; i <= shortestDistance[dest]; i++) {
        printf( format: "%d ", routeTaken[i]);
    }
    printf( format: "\n");

    numberOfNodesTraversedInDijkstra = shortestDistance[dest] + 1;

    free( Memory: shortestDistance);
    free( Memory: parent);
    return routeTaken;
}
```