스마트 모빌리티 프로그래밍

Ch 20. 자율주행에서의 이동계획, 경로탐색, 최적경로 계산, 그래프 자료구조와 알고리즘,

김 영 탁

영남대학교 기계IT대학 정보통신공학과

(Tel : +82-53-810-3940; E-mail : yse09@ynu.ac.kr)

Outline

- 자율주행에서의 이동 계획
- 그래프 자료구조와 알고리즘
- 경로탐색
- 최적 경로 계산

자율주행에서의 이동 계획,
최단거리 경로 탐색

자율 주행에서의 이동 계획 (Motion Planning)

◆ 이동 계획 (Motion Planning)

- 경로 탐색 (path finding) :
  출발지에서 목적지 까지의 경로를 탐색,
  이동 도중의 돌발 장애물에 대한 우회 경로 탐색도 포함
- 주행 제어 :
  이동 경로 상에서 조향 (steering), 속도, 가속도 등의 제어

경로 탐색 (Path Finding) 알고리즘

◆ 참고 자료

- MATLAB Tech Talk, Path Planning with A* and RRT | Autonomous Navigation, Part 4, https://www.youtube.com/watch?v=QR3U1dgc5RE

◆ Graph 기반 경로 탐색

- 출발지와 목적지가 포함된 경로 탐색 구역을 grid (격자)로 구성
- 노드 (node)와 간선 (edge)로 구성되는 그래프 자료구조 G를 준비
- 노드는 격자의 (x, y) 좌표, 현재까지 누적된 거리 정보를 포함
- 출발지 노드 (node S)를 그래프 G에 추가, 출발지 노드의 누적된 거리는 0
- 탐색 영역에서 임의 (random)로 새로운 격자를 선택
- 지금까지 선택하였던 격자 (출발지 포함)와 직선 연결할 때 장애물이 없는 지 판단하고, 만약 직선 연결이 가능하면 이 새로운 노드 (격자 좌표, 누적된 거리)와 간선 (노드-노드, 거리)을 그래프에 추가
- 새로운 격자로 목적지 노드가 선택되면, 목적지 노드와 간선을 그래프에 포함시키고, 알고리즘을 종료; 목적지에 도달하지 않았으면 다음 중간 지점을 임의로 선택하는 절차를 반복

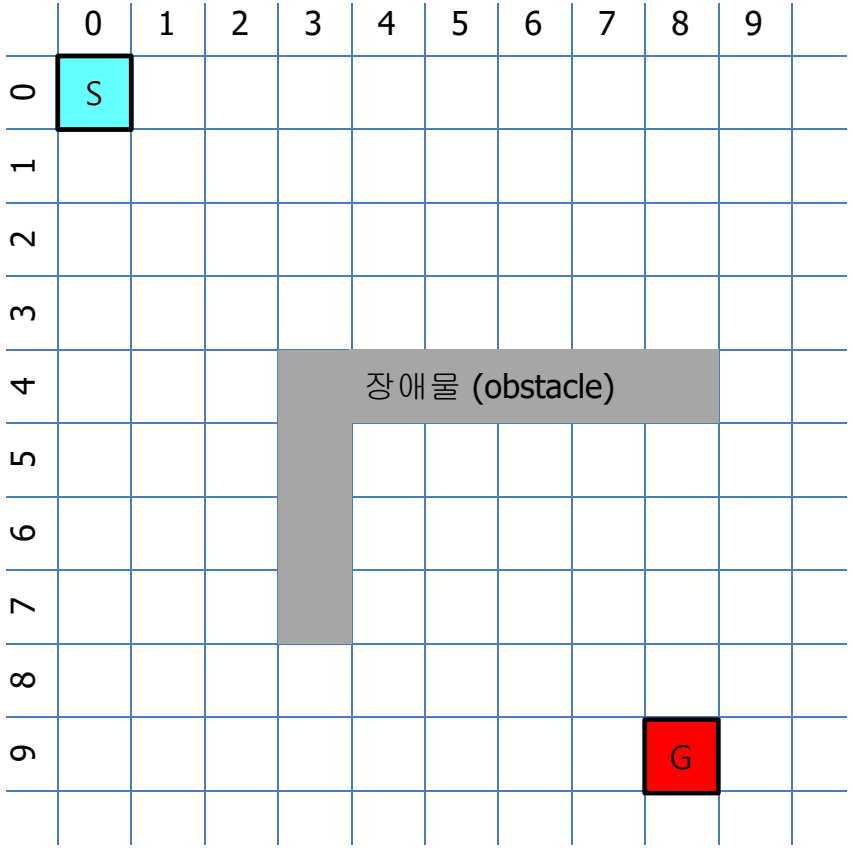
경로 탐색을 위한 자료구조

◆ Grid
- 탐색 구간 전체를 일정한 간격의 grid로 표현
- 2차원 평면에서 x, y 좌표로 특정 grid node 설정
- Grid 상에 출발지 (start)과 목적지 (goal), 장애물 (obstacle)이 좌표로 표시됨

◆ Node (또는 vertex)
- Grid 상의 특정 위치를 선택하여 노드로 표시
- grid 상의 좌표 (x, y)
- start 노드로 부터의 거리 정보
- start 노드로 부터의 경로에서 직전 노드 (prev node) 정보

◆ Weighted Edge (또는 weighted link)
- 노드와 노드를 연결하는 간선 (edge)
- 두 노드 간의 거리를 weight로 포함

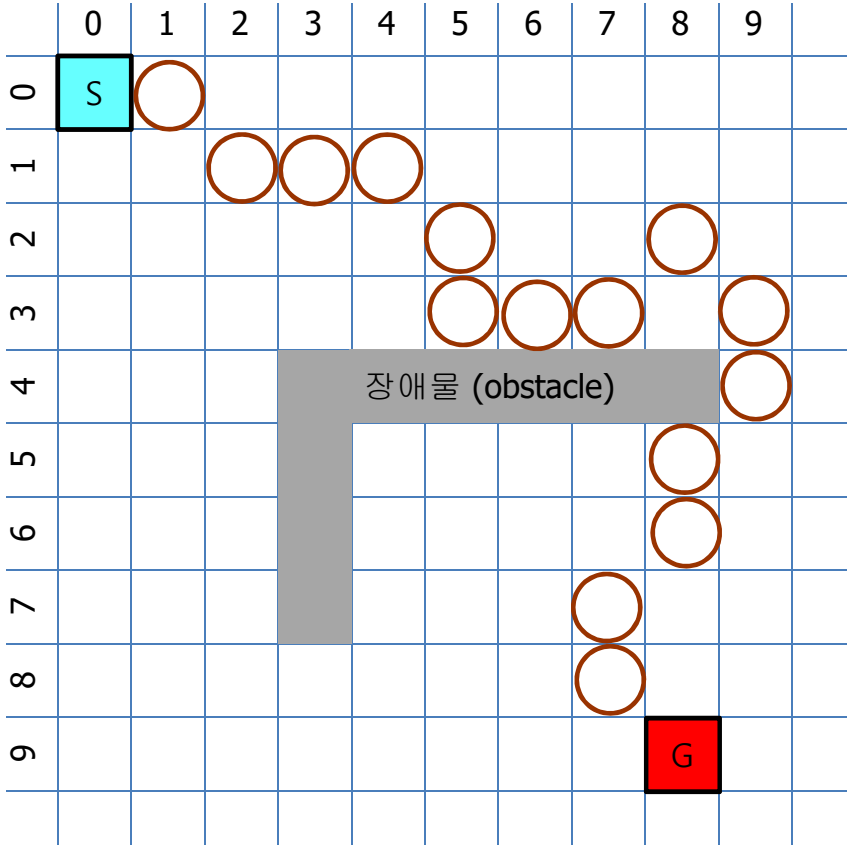
◆ Weighted Graph
- Weighted edge와 node들로 구성된 그래프



그래프 기반 경로 탐색 기법 (1) - 깊이 우선 탐색

◆ 깊이 우선 탐색 (Depth First Search)

- start node에 이웃한 이웃 grid node들을 이웃 노드 (neighbor node)로 차례로 탐색하며, Graph에 neighbor node와 edge를 graph G에 추가
- 이웃 노드의 선택은 랜덤으로 처리
- 새롭게 선택된 이웃 노드의 직전 노드를 prev node로 기록
- 이웃 노드가 목적지 (goal)이면 알고리즘 종료
- goal node로 부터 prev node를 반복적으로 찾아 역순으로 정리하면 start node로 부터 goal node까지의 경로를 파악하게 됨
- 깊이 우선 탐색으로 탐색된 경로는 최단 거리 경로를 보장하지 않음



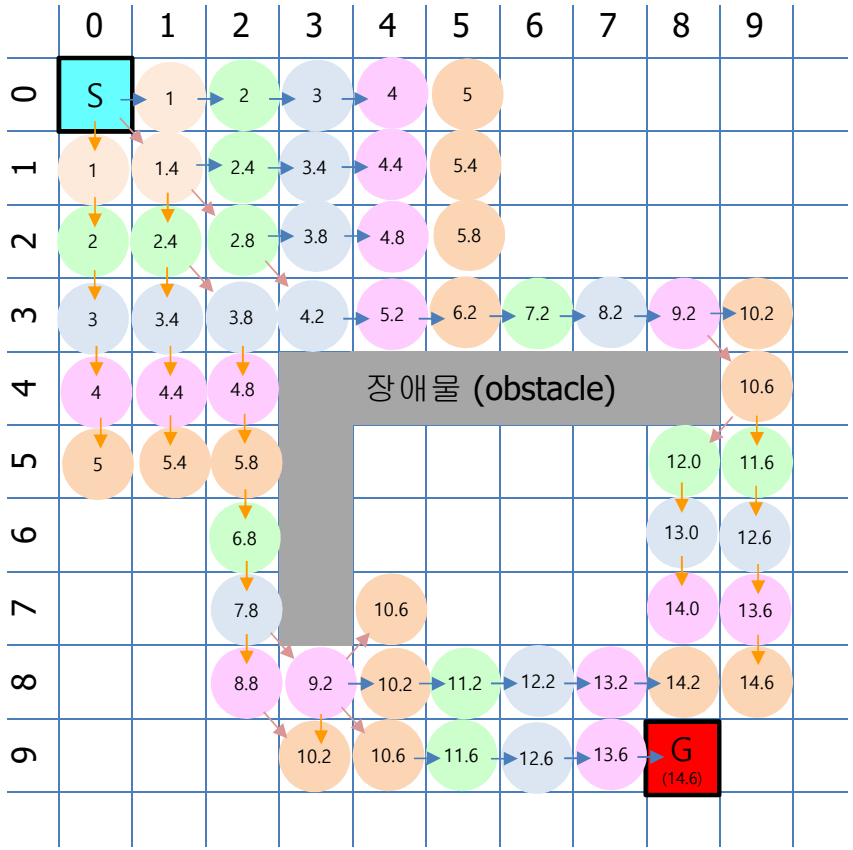
그래프 기반 경로 탐색 기법 (2) - 넓이 우선 탐색, Dijkstra

◆ 넓이 우선 탐색(Breadth First Search)

- start node로 부터 1번 만에 (대각선 포함) 도달 할 수 있는 모든 이웃 노드를 모두 차례로 탐색
- start node로 부터 2번 만에 도달할 수 있는 모든 이웃 노드들을 모두 차례로 탐색
- 목적지 (goal) 노드에 도달하면 알고리즘 종료
- 넓이 우선 탐색에서는 edge/link의 개수만을 고려

◆ Dijkstra 최단거리 탐색 (Shortest Path Search)

- 넓이 우선 탐색에서 각 edge의 거리 (distance, weight)을 고려하고, 현재까지 파악된 노드를 통하여 도달할 수 있는 가장 작은 값의 경로를 선택하며, 직전 노드를 prev node로 기록
- 출발지 부터 목적지까지의 경로 중 가장 최단 거리 경로를 찾아 줌
- 전체 grid node를 탐색하므로 시간이 많이 걸림



Dijkstra 탐색 기반 경로 탐색의 분석

◆ Dijkstra 탐색 기반 경로 탐색의 특성

- 넓이 우선 탐색에서 각 edge의 거리 (distance, weight)을 고려하고, 현재까지 파악된 노드를 통하여 도달할 수 있는 가장 작은 값의 경로를 선택하며, 직전 노드를 prev node로 기록
- 출발지부터 목적지까지의 경로 중 가장 최단 거리 경로를 찾아 줌
- 모든 grid node를 탐색하므로 시간이 많이 걸림

◆ 경로 탐색 기법의 성능 개선

- 모든 grid node를 탐색하지 않고, random으로 grid node를 선정한 후, 이 random으로 선택된 grid node에서 장애물을 거치지 않는 가장 이웃하는 노드를 찾아 연결
- random으로 grid node를 선택할 때 목적지에 가까운 grid node를 선택할 확률을 높여서 선택

그래프 자료구조와 알고리즘

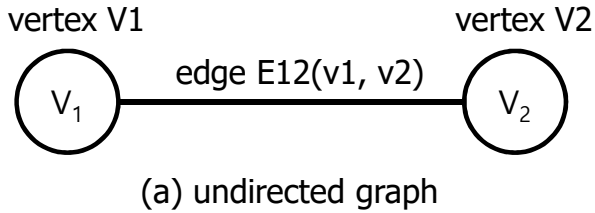
Vertex/Node, Edge

◆ 정점 (Vertex), 노드(Node)

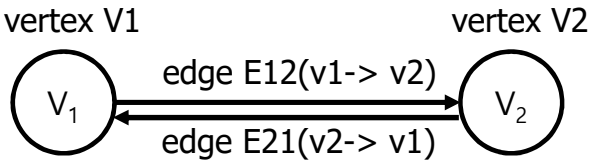
- 지도상의 도시 (city), 공항
- 연관성 분석 대상의 정보

◆ 간선 (Edge)

- 도시를 연결하는 도로, 항공편
- 분석 대상 정보간의 연관성
- 방향성 간선 또는 무방향성 간선으로 표현

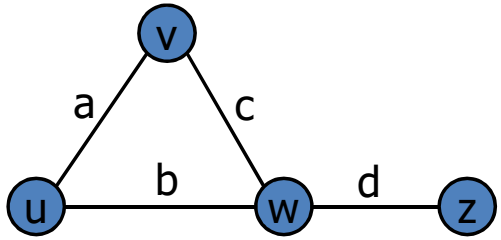


(a) undirected graph



(b) directed graph

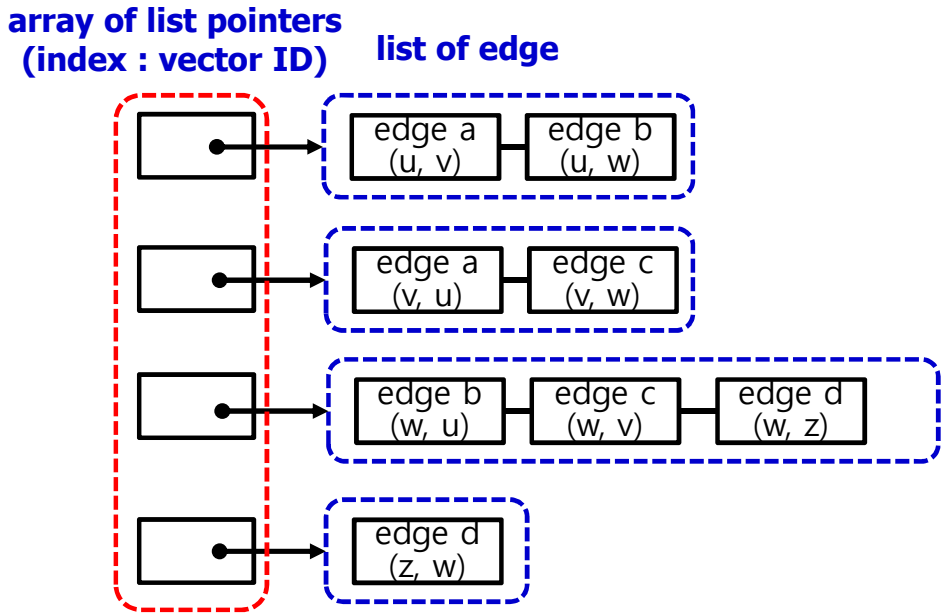
그래프의 연결성 표현



(a) 그래프 토폴로지 **(Graph Topology)**

Vertex (u, 0)
Vertex (v, 1)
Vertex (w, 2)
Vertex (z, 3)

(b) 정점 배열 **(Vertex Array)**



(c) 인접리스트 배열 **(Adjacency List Array)**

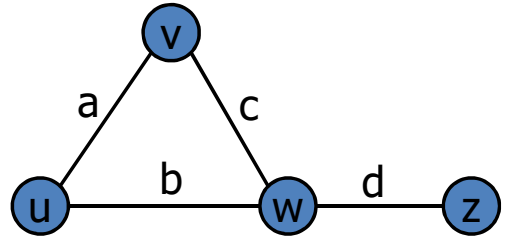
인접행렬 (Adjacency Matrix), 거리표

◆ 2차원 인접 행렬 (2-D adjacency matrix)

- 각 노드간의 간선 정보를 표로 정리

◆ 거리표 (Distance Table)

- 만약 두 노드를 직접 연결하는 간선이 없는 경우: ∞
- 출발지와 도착지가 동일 노드인 경우: 0
- 두 노드간에 간선이 있는 경우 : 간선의 가중치



	u	v	w	z
u	0	a	b	∞
v	a	0	c	∞
w	b	c	0	d
z	∞	∞	d	0

그래프 탐색 (Graph Search)

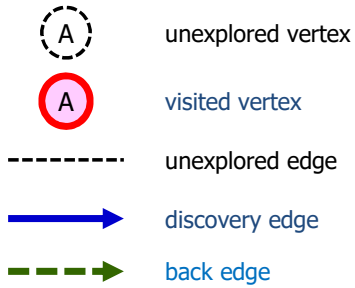
◆ 깊이 우선 탐색 (Depth First Search)

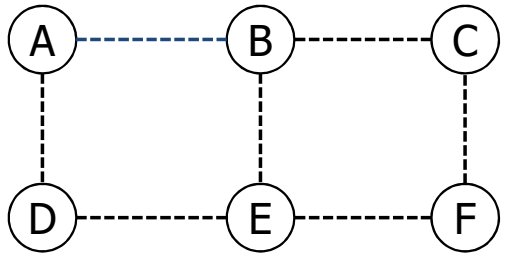
- 시작 노드 (start node)로 부터 간선이 연결되어 있는 새로운 노드를 찾아 깊이 우선으로 탐색
- 목적지까지의 경로가 존재하는지를 파악하는 것에 중점
- 탐색된 결과의 경로가 최단 거리 경로를 보장하지 않음

◆ 넓이 우선 탐색 (Breadth First Search)

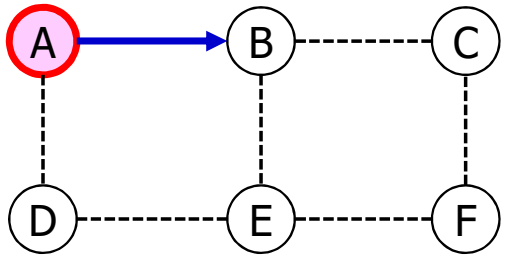
- 현재 방문중인 노드에 연결되어 있는 모든 노드들을 확인한 후, 다음 레벨의 노드들을 탐색
- 레벨 0: 시작노드
- 레벨 1: 시작노드로 부터 하나의 간선을 통하여 이동할 수 있는 노드 그룹
- 레벨 2: 시작노드로 부터 두 개의 간선을 통하여 이동할 수 있는 노드 그룹
- 넓이우선탐색으로 탐색된 경로는 간선의 개수가 최소가 되는 경로

그래프의 깊이우선탐색 (Depth First Search) (1)

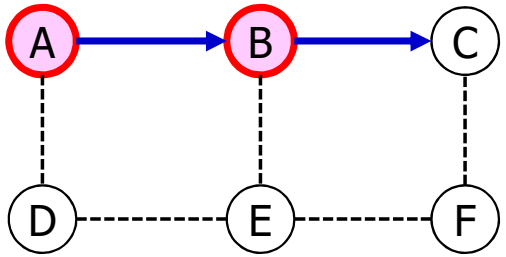




(a) Graph to be searched

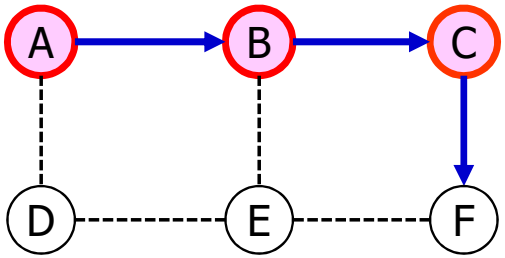


(b) Vertex A selected,
Edge (A-B) searched

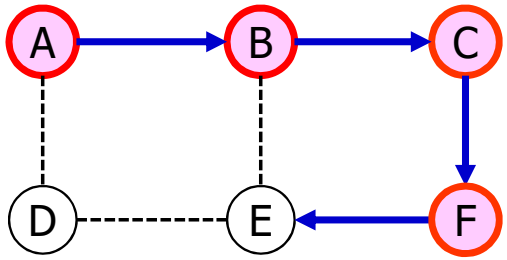


(c) Vertex B selected,
Edge (B-C) searched

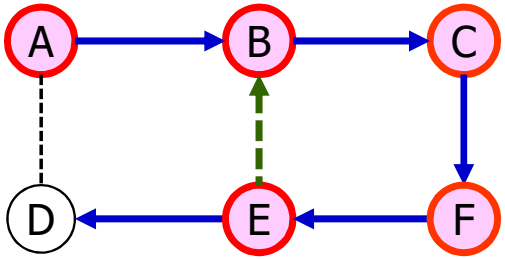
그래프의 깊이우선탐색 (Depth First Search) (2)



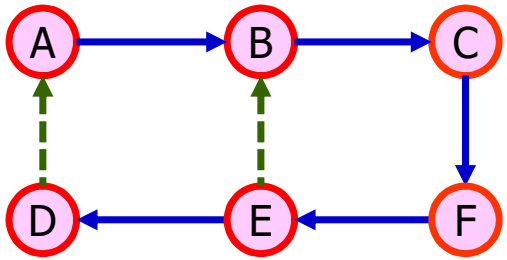
(d) Vertex C selected,
Edge (C-F) searched



(e) Vertex F selected,
Edge (F-E) searched

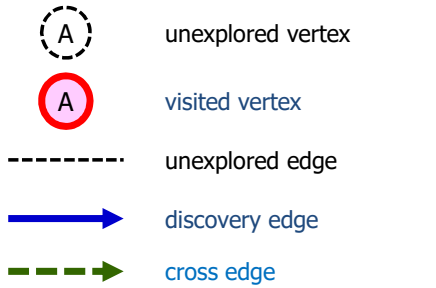


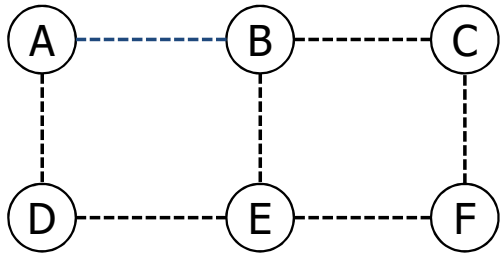
(f) Vertex E selected,
Edge (B-E) and Edge (E-D) searched



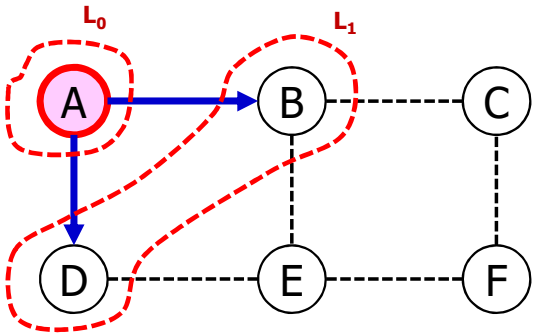
(g) Vertex D selected,
Edge (D-A) searched

그래프의 넓이우선탐색 (Breadth First Search) (1)

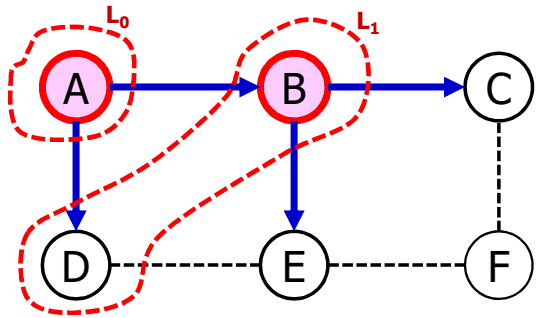




(a) Graph to be searched

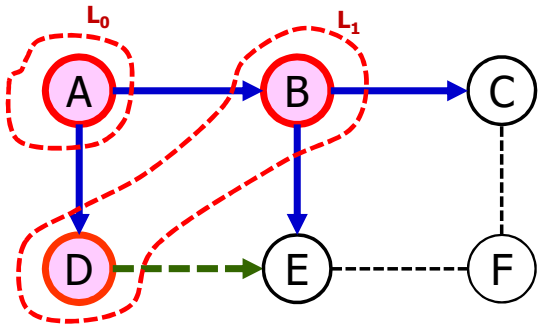


(b) Vertex A selected,
Edges (A-B), (A_D) searched

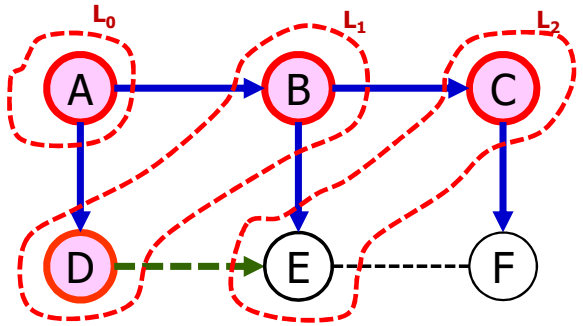


(c) Vertex B selected,
Edges (B-C), (B-E) searched

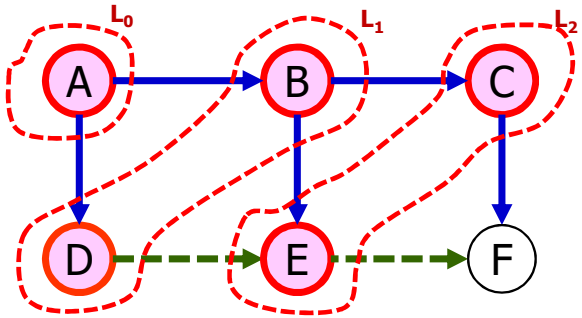
그래프의 넓이우선탐색(Breadth First Search) (2)



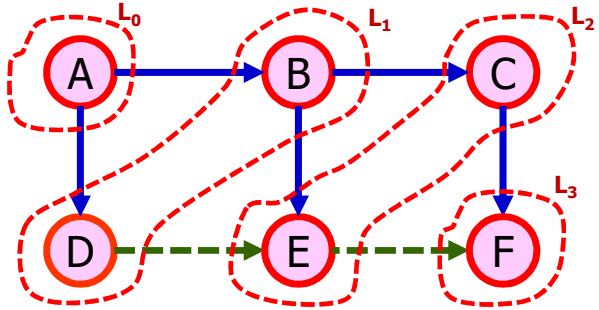
(d) Vertex D selected,
Edges (D-E) searched



(e) Vertex C selected,
Edges (C-F) searched



(f) Vertex E selected,
Edges (B-C), (B-E), (D-E) searched



(g) Vertex F selected

가중치 그래프 (weighted graph)에서 최단거리 경로찾기

◆ 가중치 그래프 (weighted graph)

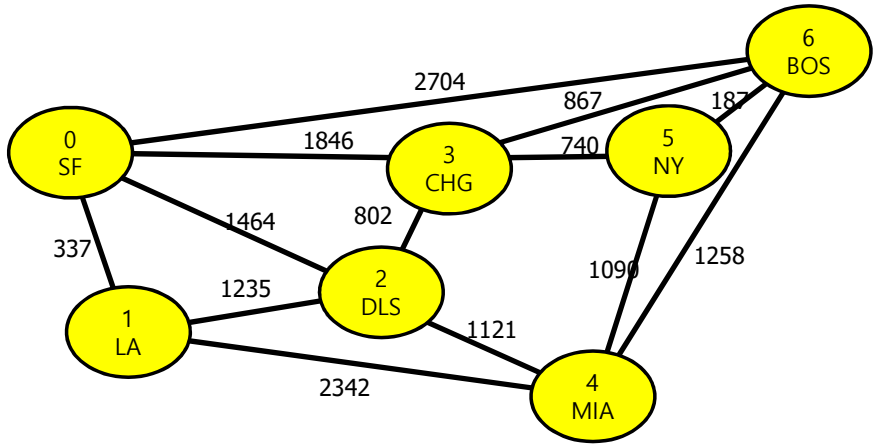
- 그래프의 간선에 가중치 (거리, 시간, 비용)가 설정됨

◆ 최단거리 경로 탐색 (shorted distance path)

- 가중치 그래프 상에서 지정된 두 정점간의 경로 중 최단 거리 경로의 누적된 거리가 최소가 되는 경로

◆ 주요 응용 분야

- 자동차 네비게이션에서 최단 거리 경로 탐색
- 인터넷에서의 경로 설정
- 항공 여행에서의 항공편 설정



다익스트라 알고리즘 (Dijkstra's Algorithm)

◆ Dijkstra's Algorithm

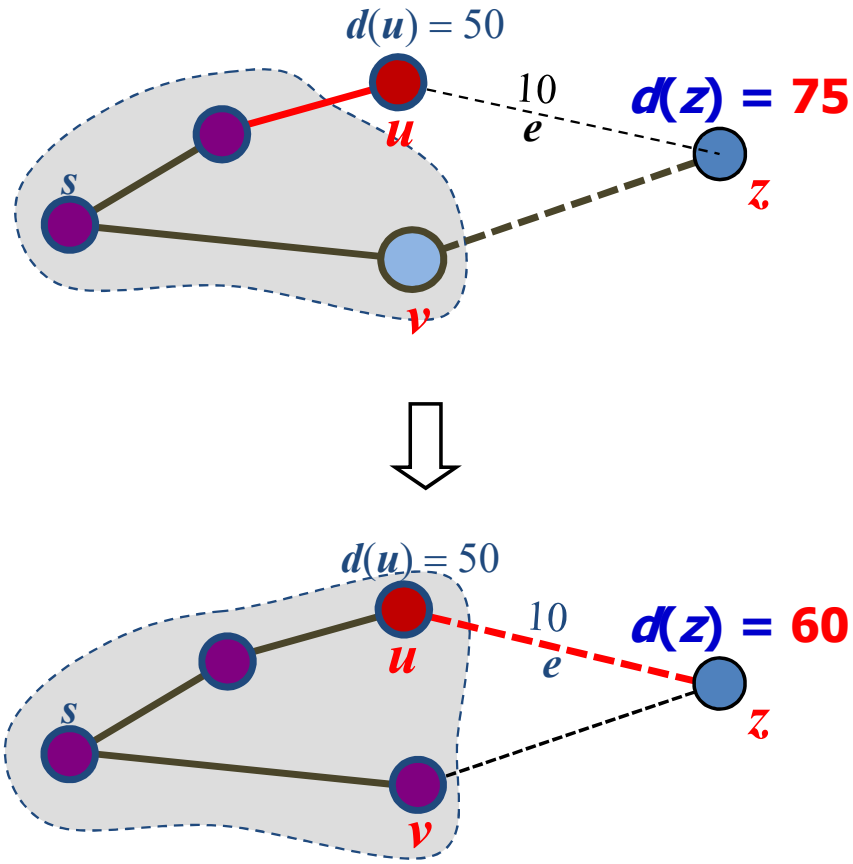
- 그래프의 간선들에 가중치 (weight)가 설정된 그래프에서 최단거리 경로 탐색

Algorithm Dijkstra_Shortest_Path(G, start, target)
전달인수: 가중치가 할당된 그래프 G, 시작노드 start, 목적지노드 target
반환값: 시작노드에서 목적지 노드까지의 최단 거리 경로
1: 시작노드를 기준으로 거리표 (distance table)의 초기화
2: 모든 노드를 "Not_Selected"상태로 설정
3: 시작노드를 "Selected(선택)"로 설정
4: 시작노드로부터 "Not_Selected(비선택)" 노드까지의 누적거리를 계산
5: while (선택되지 않은 노드가 없을 때까지):
6: 현재 "Not_Selected" 노드 중 누적거리가 가장 짧은 노드를 선택하여 "Selected"로 설정
7: 만약 새롭게 선택된 노드가 목적지 노드이면 현재까지의 경로와 누적 거리를 반환하여 알고리즘 종료
8: 새롭게 선택된 노드를 경유하여 도달할 수 있는 노드의 누적 거리를 계산하여, 기존 경로보다 누적 비용이 더 작은 경우 경로 갱신 (Edge relaxation)

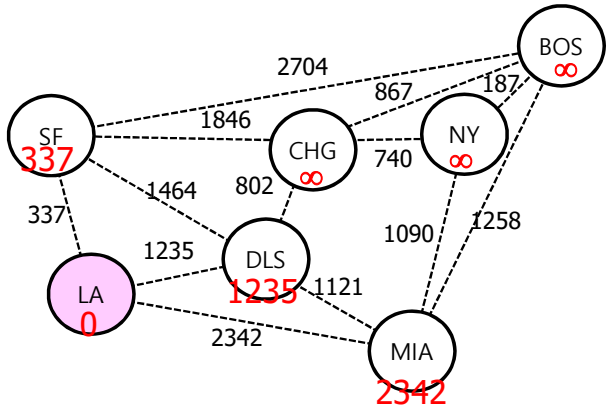
Edge Relaxation

- **Dijstra 알고리즘에서 새롭게 선택된 노드 u의 간선 *e* = (*u*, *z*)에 연결된 노드 z**
  - *u* 는 새롭게 선택된 노드
  - *z* 는 아직 선택되지 않은 노드
- **간선 e를 통한 노드 z의 누적 거리가 더 좋은 조건인 경우, 경로 갱신 :**

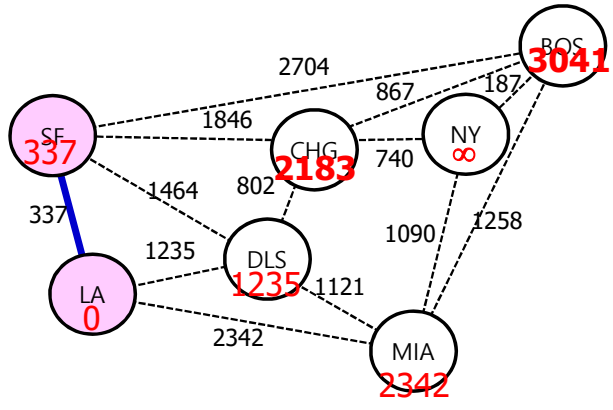
$$d(z) \leftarrow \min\{ d(z),\ d(u) + weight(e) \}$$



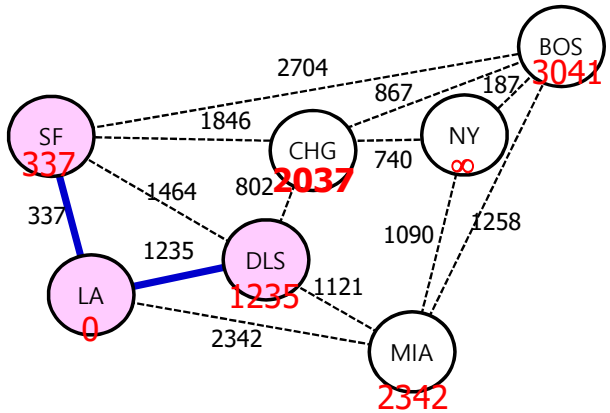
가중치가 설정된 그래프에서의 Dijkstra 알고리즘 실행 예 (1)



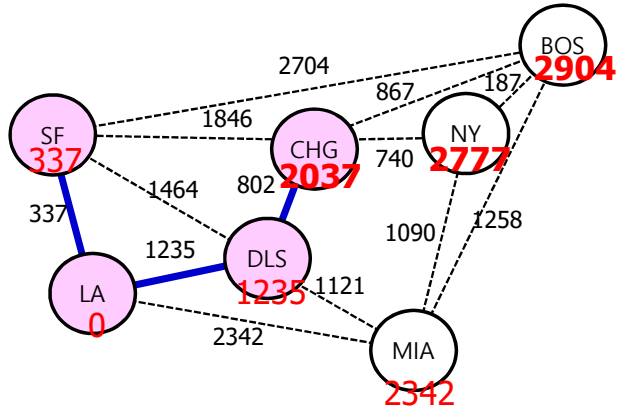
(a) round 0



(b) round 1

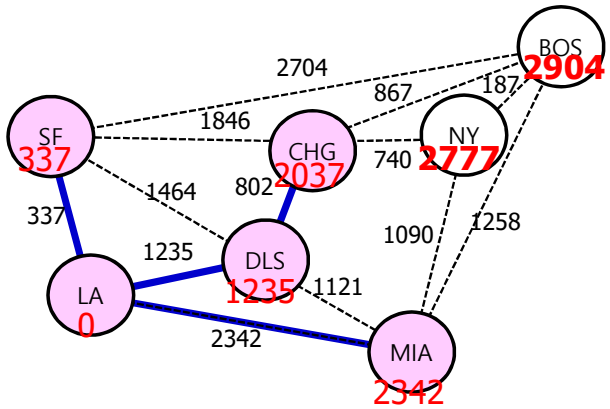


(c) round 2

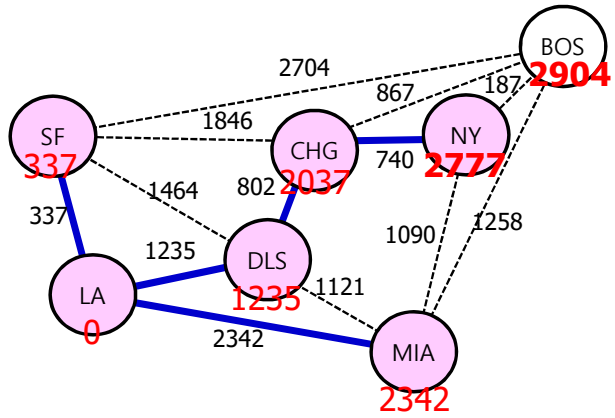


(d) round 3

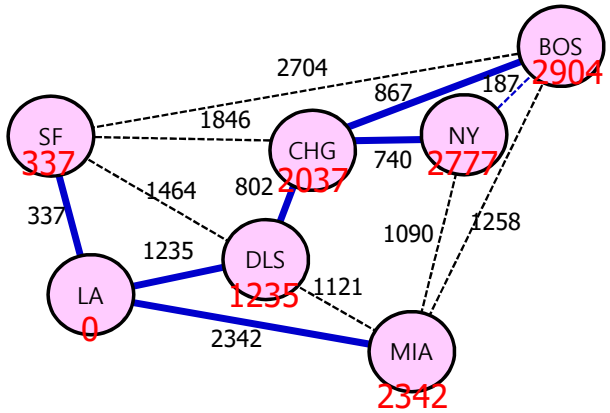
가중치가 설정된 그래프에서의 Dijkstra 알고리즘 실행 예 (2)



(e) round 4



(f) round 5



(g) round 6

그래프 자료구조와 알고리즘의 구현

그래프 알고리즘 구현

```
# user-defined module - MyGraph.py  (1)
import sys # for sys.maxsize as PLUS_INF

class Node(object):
    def __init__(self, name):
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class WeightedEdge(Edge):
    def __init__(self, src_nm, dest_nm, weight=1.0):
        self.src_nm = src_nm
        self.dest_nm = dest_nm
        self.weight = weight
    def getSource_nm(self):
        return self.src_nm
    def getDestination_nm(self):
        return self.dest_nm
    def getWeight(self):
        return self.weight
    def __str__(self):
        return "{:3}->({:3})->{}".format(self.src_nm, self.weight, self.dest_nm)
```

```
# user-defined module - MyGraph.py  (2)

class WeightedGraph(object):
    def __init__(self):
        self.nodes = [] # list of Node(v_id, v_names)
        self.node_names = []
        self.wedges = []  # list of weighted_edges
        self.adjacencyList = {} # dict of {src_nm:list of node_names)
        self.edgeWeights = {} # dictionary of {edge(src_nm, dest_nm):weight}
    def addNode(self, node):
        if node in self.nodes:
            raise ValueError("Duplicated node")
        else:
            self.nodes.append(node)
            node_nm = node.getName()
            self.node_names.append(node_nm)
            self.adjacencyList[node_nm] = []
    def addEdge(self, weighted_edge):
        src_nm = weighted_edge.getSource_nm()
        dest_nm = weighted_edge.getDestination_nm()
        if not (src_nm in self.node_names and dest_nm in self.node_names):
            raise ValueError("Node not in graph")
        self.wedges.append(weighted_edge)
        self.adjacencyList[src_nm].append(dest_nm)
        self.edgeWeights[(src_nm, dest_nm)] = weighted_edge.getWeight()
    def getNeighbors(self, node_nm):
        #print(" WeightedGraph::getNeighbors({}) = {}".format(node_nm, self.adjacencyList[node_nm]))
        return self.adjacencyList[node_nm]
    def getAdjacencyList(self):
        return self.adjacencyList
    def getNode_NMs(self):
        return self.node_names
```

user-defined module - MyGraph.py (3)

```
    def getWEdges(self):
        return self.wedges

    def getEdgeWeight(self, edge):
        if (edge.src_nm, edge.dest_nm) in self.edgeWeights:
            return self.edgeWeights[(edge.src_nm, edge.dest_nm)]
        else:
            None

    def printConnectivity(self):
        for node_nm in self.node_names:
            print(" AdjacencyList[{}] = {}"\
              .format(node_nm, self.adjacencyList[node_nm]))

    def printEdges(self):
        eCount = 0
        for e in self.wedges:
            print("  {}".format(e), end=', ')
            eCount += 1
            if eCount % 5 == 0:
                print()

    def __str__(self):
        result = ''
        for src in self.nodes:
            for dest in self.edges[src]:
                result = result + src.getName() + '->'\
                        + dest.getName() + '\n'
                return result[:-1] # omit final newline
```

user-defined module - MyGraph.py (4)

```
    def printConnectivity(self):
        print("Inter-city distance table")
        s = " " * 5 + "|"
        for city in self.node_names:
            s += "{:>5s}".format(city)
        s += "\n" + "-" * 5 + "+"
        for i in range(len(self.node_names)):
            s += "-" * 5
        s += "\n"
        for i in range(len(self.node_names)):
            s += "{:^5s}|".format(self.node_names[i])
            for j in range(len(self.node_names)):
                if (self.node_names[i] == self.node_names[j]):
                    s += "{:5d}".format(int(0))
                    continue
                dist = self.getEWeight(self.node_names[i],
                    self.node_names[j])
                if (dist == None) and
                    (self.node_names[i] != self.node_names[j]):
                    s += "{:>5s}".format("oo")
                else:
                    s += "{:5d}".format(dist)
            s += "\n"
        print(s)
```

```
# user-defined module - MyGraph.py (5)

def fgetWeightedGraph(file_name):
    G = WeightedGraph()
    node_names = []
    w_edges = []
    fin = open(file_name, "r")
    first_line = fin.readline()
    gname, num_nodes_str = first_line.split()
    for icd in fin.readlines():
        (c1_nm, c2_nm, dist_str) = icd.split()
        dist = int(dist_str)
        if c1_nm not in node_names:
            node_names.append(c1_nm)
        if c2_nm not in node_names:
            node_names.append(c2_nm)
        w_edges.append(WeightedEdge(c1_nm, c2_nm, dist))
        w_edges.append(WeightedEdge(c2_nm, c1_nm, dist))
    fin.close()

    print("fget_Graph() - adding nodes into Graph : ", end='')
    for i in range(len(node_names)):
        v_name = node_names[i]
        node = Node(v_name)
        print("{}, ".format(node.getName()), end='')
        G.addNode(node)
    print()

    for we in w_edges:
        #print("\ninitGraph() :: adding weighted_edge {} into Graph".format(we))
        G.addEdge(we)
    return G
```

```
# user-defined module - MyGraph.py (6)

def printWeightedGraph(wg): # print weighted graph
    node_names = wg.getNode_NMs()
    print("printWeightedGraph() :: Nodes : ", node_names)
    edges = wg.getWEdges()
    print("printWeightedGraph() :: WeightedEdges :")
    eCount = 0
    for e in edges:
        print("  {}".format(e), end=', ')
        eCount += 1
        if eCount % 5 == 0:
            print()
    print()

    print("\nprintWeightedGraph() :: Connectivity/distance Table:")
    wg.printConnectivity()

def printPath(path):
    result = ''
    for i in range(len(path)):
        result += str(path[i])
        if i != len(path) -1:
            result += '->'
    return result
```

```
# user-defined module - MyGraph.py (7)

def DFS(graph, begin_nm, end_nm, path, shortest): # depth first search
    #print("DFS:: begin({}), end({}))".format(begin_nm, end_nm))
    path.append(begin_nm)
    #print("Current DFS path : ", printPath(path))
    if begin_nm == end_nm:
        return path
    for node_nm in graph.getNeighbors(begin_nm):
        if node_nm not in path: # avoid cycle
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node_nm, end_nm, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest
```

```
# user-defined module - MyGraph.py (8)

def BFS(graph, start_nm, end_nm): # breadth first search
    initPath = [start_nm]
    pathQueue = [initPath]
    #count = 0
    while len(pathQueue) != 0:
        #print("Round ({:2d}) - pathQueue : ".format(count), end=' ')
        """
        for path in pathQueue:
            print("{}".format(printPath(path)), end=' ')
        print()
        """
        tmpPath = pathQueue.pop(0)
        #print(" - current tmpPath: {}".format(printPath(tmpPath)))
        lastNode_nm = tmpPath[-1]
        if lastNode_nm == end_nm:
            return tmpPath
        for nextNode_nm in graph.getNeighbors(lastNode_nm):
            if nextNode_nm not in tmpPath:
                newPath = tmpPath + [nextNode_nm]
                pathQueue.append(newPath)
        #count += 1
    return None
```

```
# user-defined module - MyGraph.py (9)

PLUS_INF = sys.maxsize # define as max of integer
def Dijkstra(G, start_nm, end_nm): # Dijkstra shortest path
    errorInLoop = False
    nodeAccWeight= {} # dictionary of node:accumulated_weight_from_start
    nodeStatus = {}
    prevNodes_nm = {} # previous node in the path from the start to the end
    selectedNodes = []
    remainingNodes = []
    wEdges = G.getWEdges()
    #print("Dijkstra::edges : ", edges)
    for node_nm in G.node_names:
        e = Edge(start_nm, node_nm)
        if node_nm == start_nm:
            eWeight = 0
        else:
            eWeight = G.getEdgeWeight(e)
            if eWeight == None:
                eWeight = PLUS_INF
        print(" Initial weight of edge ({}) = {}: ".format(e, eWeight))
        nodeAccWeight[node_nm] = eWeight
        nodeStatus[node_nm] = False # not selected yet
        prevNodes_nm[node_nm] = start_nm
        if node_nm != start_nm:
            remainingNodes.append(node_nm)
    nodeAccWeight[start_nm] = 0
    nodeStatus[start_nm] = True
    selectedNodes.append(start_nm)
    #print("nodeAccWeight : ", nodeAccWeight)
    #print("Initial status of prevNode : ", prevNodes_nm)
```

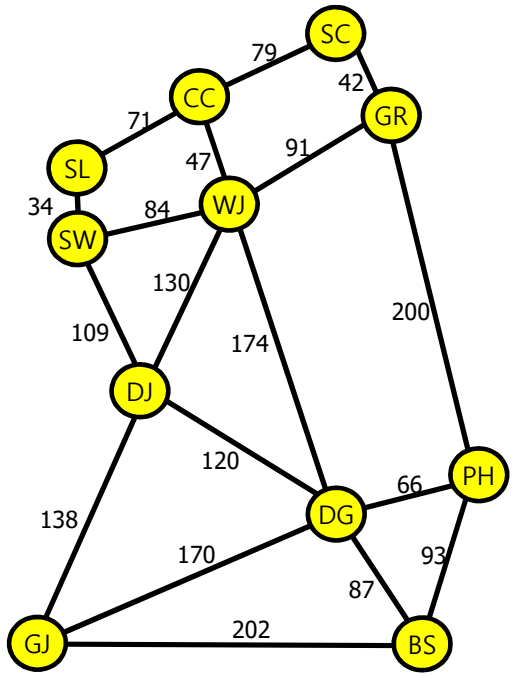
```
# user-defined module - MyGraph.py (10)

    count = 1
    while len(remainingNodes) != 0:
        #print(">>> Round {} :".format(count))
        minAccWeight = PLUS_INF
        minNode = None
        #print(" -- currently selected {}, remaining {}“\
            .format(selectedNodes, remainingNodes))
        for n in remainingNodes:
            nAccWeight = nodeAccWeight[n]
            #print(" -- evaluating node ({:3}), nAccWeight({}) ...".format(n, nAccWeight))
            #print(" -- current minAccWeight = ", minAccWeight)
            if nAccWeight != None and nAccWeight < minAccWeight:
                minNode, minAccWeight = n, nodeAccWeight[n]
                #print("    ==> minAccWeight updated by newMinNode ({}) with minAccWeight
                  ({})".format(minNode, minAccWeight))
        if minNode == None:
            print("No minNode was selected at this round !!")
            print("Error - graph is not fully connected !!")
            errorInLoop = True
            break
        else:
            #print(" -- newly selected minNode : {}".format(minNode))
            selectedNodes.append(minNode)
            minAccWeight = nodeAccWeight[minNode]
            # edge relaxations
            for rn in remainingNodes:
                if rn == minNode:
                    continue
                e = Edge(minNode, rn)
                eWeight = G.getEdgeWeight(e)
                #print(" -- eWeight({}->{}):{}".format(minNode, rn, eWeight))
                if eWeight == None:
                    continue
```

```
# user-defined module - MyGraph.py (11)

            if nodeAccWeight[rn] > minAccWeight + eWeight:
                nodeAccWeight[rn] = minAccWeight + eWeight
                prevNodes_nm[rn] = minNode
                #print(" -- Updated nodeAccWeight for node ({:3}) with prevNode
                    ({:3})".format(rn, minNode))
        if minNode == end_nm:
            # reached to destination
            break
        remainingNodes.remove(minNode)
    #print(" -- PrevNode : ", prevNodes_nm)
    #print(" -- Remaining nodes : ", end='')
    """
    for rn in remainingNodes:
        print("{} ({})".format(rn,nodeAccWeight[rn]), end=', ')
    print()
    """
    count += 1
#
if errorInLoop == True:
    return None

print(" prevNode : ", prevNodes_nm)
path = [end_nm]
cur_node_nm = end_nm
while cur_node_nm in selectedNodes:
    #print("Current path : ", path)
    if cur_node_nm == start_nm:
        break
    else:
        cur_node_nm = prevNodes_nm[cur_node_nm]
        path.insert(0,cur_node_nm)
        #print("Current path : ", path)
return path, nodeAccWeight[end_nm]
```

```
# Graph with DFS, BFS, Dijkstra, MST(MinimumSpanningTree) (5)

def initGraph(G):
    node_names = ["SF", "LA", "DLS", "CHG", "MIA", "NY", "BOS"]
    w_edges = [WeightedEdge("SF", "LA", 337), WeightedEdge("LA", "SF", 337),\
               WeightedEdge("SF", "DLS", 1464), WeightedEdge("DLS", "SF", 1464),\
               WeightedEdge("SF", "BOS", 2704), WeightedEdge("BOS", "SF", 2704),\
               WeightedEdge("SF", "CHG", 1846), WeightedEdge("CHG", "SF", 1846),\
               WeightedEdge("LA", "DLS", 1235), WeightedEdge("DLS", "LA", 1235),\
               WeightedEdge("LA", "MIA", 2342), WeightedEdge("MIA", "LA", 2342),\
               WeightedEdge("DLS", "MIA", 1121), WeightedEdge("MIA", "DLS", 1121),\
               WeightedEdge("DLS", "CHG", 802), WeightedEdge("CHG", "DLS", 802),\
               WeightedEdge("CHG", "NY", 740), WeightedEdge("NY", "CHG", 740),\
               WeightedEdge("CHG", "BOS", 867), WeightedEdge("BOS", "CHG", 867),\
               WeightedEdge("NY", "MIA", 1090), WeightedEdge("MIA", "NY", 1090),\
               WeightedEdge("NY", "BOS", 187), WeightedEdge("BOS", "NY", 187),\
               WeightedEdge("BOS", "MIA", 1258), WeightedEdge("MIA", "BOS", 1258) ]
    for i in range(len(node_names)):
        v_name = node_names[i]
        node = Node(v_name)
        print("initGraph() :: adding node ({}) into Graph".format(node.getName()))
        G.addNode(node)
    for we in w_edges:
        #print("\ninitGraph() :: adding weighted_edge {} into Graph".format(we))
        G.addEdge(we)
    return G

def searchSP_DFS(graph, start_nm, end_nm):
    return DFS(graph, start_nm, end_nm, [], None)

def searchSP_BFS(graph, start_nm, end_nm):
    return BFS(graph, start_nm, end_nm)

def searchSP_Dijkstra(graph, start_nm, end_nm):
    return Dijkstra(graph, start_nm, end_nm)
```

Application of WeightedGraph, Dijkstra's Shortest Path

```
#  Application of WeightedGraph with Dijkstra Shortest Path First Search
from MyGraph import *

EdgesPerLine = 5
if __name__ == "__main__":
    WG = fgetWeightedGraph("KR_11_cities.txt")
    printWeightedGraph(WG)
    start_nm = "GJ" #
    end_nm = "SC" #
    print("Trying ShortestPath_Dijkstra : ({} -> {})".format(start_nm, end_nm))
    path_Dijkstra, path_cost = Dijkstra(WG, start_nm, end_nm)
    print("Found shortestPath_Dijkstra ({} -> {}): {}, total_path_cost ={}"\
        .format(start_nm, end_nm, path_Dijkstra, path_cost))

    start_nm = "SC"
    end_nm = "GJ"
    print("Trying ShortestPath_Dijkstra : ({} -> {})".format(start_nm, end_nm))
    path_Dijkstra, path_cost = Dijkstra(WG, start_nm, end_nm)
    print("Found shortestPath_Dijkstra ({} -> {}): {}, total_path_cost ={}"\
        .format(start_nm, end_nm, path_Dijkstra, path_cost))
```



KR_InterCityDist - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

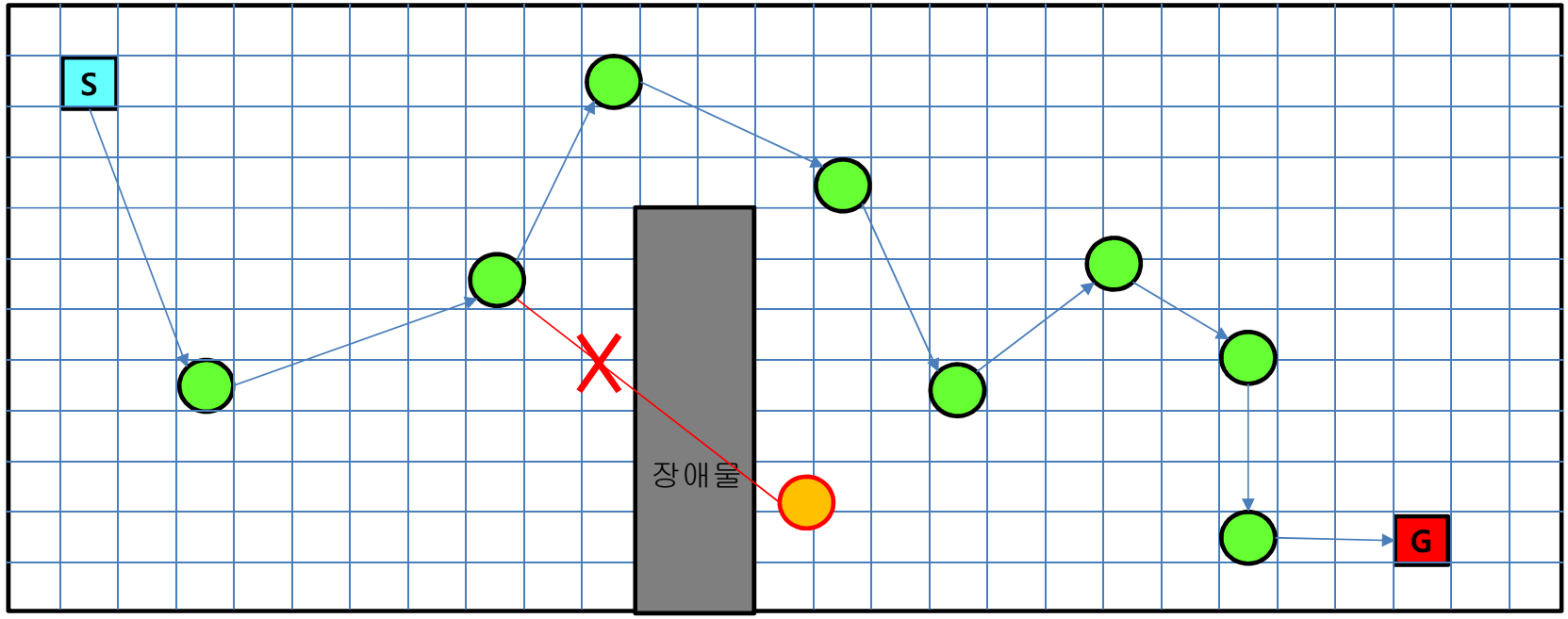
```
SL CC 71
CC SC 79
SL SW 34
CC WJ 47
SC GR 42
SW WJ 84
WJ GR 91
SW DJ 109
WJ DJ 130
WJ DG 174
GR PH 200
DJ GJ 138
DJ DG 120
GJ DG 170
DG PH 66
DG BS 87
GJ BS 202
PH BS 93
```

```
fget_Graph() - adding nodes into Graph : SL, CC, SC, SW, WJ, GR, DJ, DG, PH, GJ, BS,
printWeightedGraph() :: Nodes :  ['SL', 'CC', 'SC', 'SW', 'WJ', 'GR', 'DJ', 'DG', 'PH', 'GJ', 'BS']
printWeightedGraph() :: WeightedEdges :
  (SL->CC: 71),   (CC->SL: 71),   (CC->SC: 79),   (SC->CC: 79),   (SL->SW: 34),
  (SW->SL: 34),   (CC->WJ: 47),   (WJ->CC: 47),   (SC->GR: 42),   (GR->SC: 42),
  (SW->WJ: 84),   (WJ->SW: 84),   (WJ->GR: 91),   (GR->WJ: 91),   (SW->DJ:109),
  (DJ->SW:109),   (WJ->DJ:130),   (DJ->WJ:130),   (WJ->DG:174),   (DG->WJ:174),
  (GR->PH:200),   (PH->GR:200),   (DJ->GJ:138),   (GJ->DJ:138),   (DJ->DG:120),
  (DG->DJ:120),   (GJ->DG:170),   (DG->GJ:170),   (DG->PH: 66),   (PH->DG: 66),
  (DG->BS: 87),   (BS->DG: 87),   (GJ->BS:202),   (BS->GJ:202),   (PH->BS: 93),
  (BS->PH: 93),

printWeightedGraph() :: Connectivity/distance Table:
Inter-city distance table
     |   SL   CC   SC   SW   WJ   GR   DJ   DG   PH   GJ   BS
-----+---------------------------------------------------------
 SL  |    0   71   oo   34   oo   oo   oo   oo   oo   oo   oo
 CC  |   71    0   79   oo   47   oo   oo   oo   oo   oo   oo
 SC  |   oo   79    0   oo   oo   42   oo   oo   oo   oo   oo
 SW  |   34   oo   oo    0   84   oo  109   oo   oo   oo   oo
 WJ  |   oo   47   oo   84    0   91  130  174   oo   oo   oo
 GR  |   oo   oo   42   oo   91    0   oo   oo  200   oo   oo
 DJ  |   oo   oo   oo  109  130   oo    0  120   oo  138   oo
 DG  |   oo   oo   oo   oo  174   oo  120    0   66  170   87
 PH  |   oo   oo   oo   oo   oo  200   oo   66    0   oo   93
 GJ  |   oo   oo   oo   oo   oo   oo  138  170   oo    0  202
 BS  |   oo   oo   oo   oo   oo   oo   oo   87   93  202    0

Trying ShortestPath_Dijkstra : (GJ -> SC)
Found shortestPath_Dijkstra (GJ -> SC): ['GJ', 'DJ', 'WJ', 'CC', 'SC'], total_path_cost =394
Trying ShortestPath_Dijkstra : (SC -> GJ)
Found shortestPath_Dijkstra (SC -> GJ): ['SC', 'CC', 'WJ', 'DJ', 'GJ'], total_path_cost =394
```

경로 탐색 (Path Finding) 알고리즘

Graph 기반 경로 탐색 (1) - Random Walk

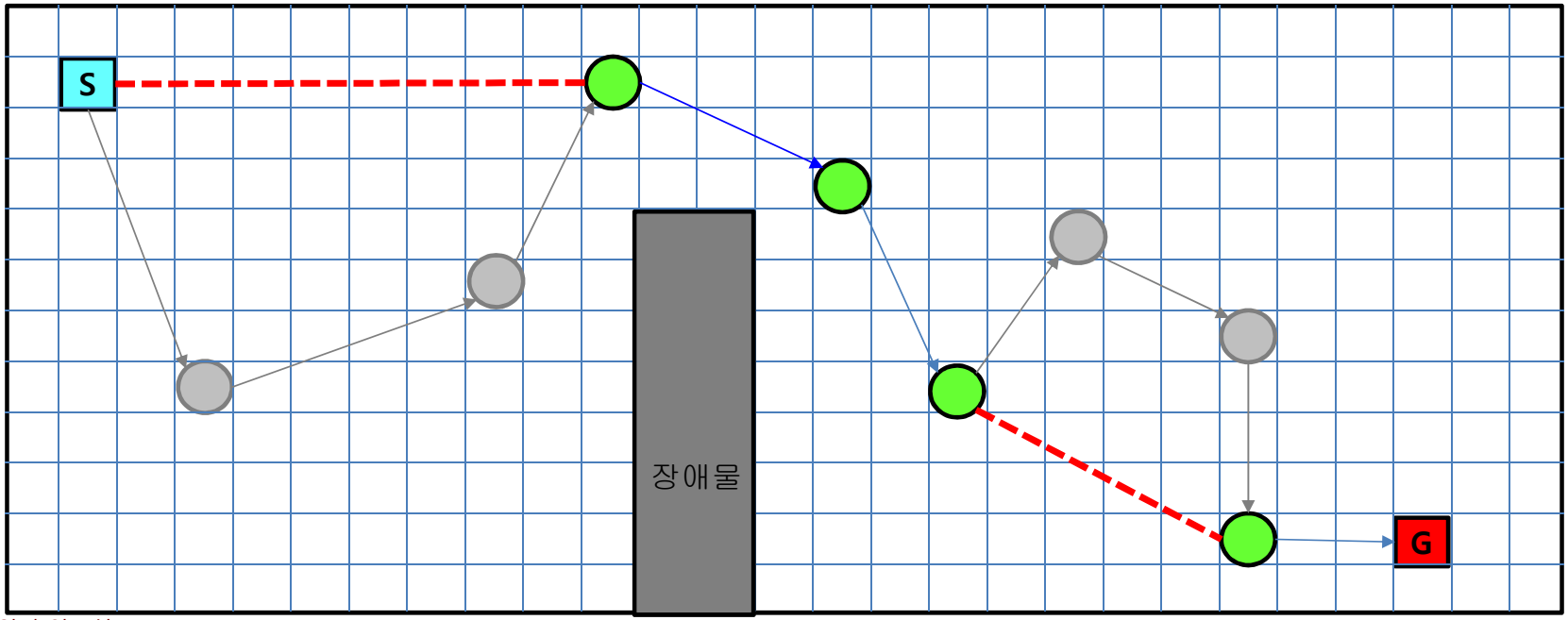
◆ Graph 기반 경로 탐색 - Random Walk



Graph 기반 경로 탐색 (2) - Random Walk의 개선

◆ Random Walk의 경로 개선

- 인접된 노드 간의 경로에서 단축 가능성을 확인하고, 경로 재설정 : edge relaxation
- 임의 노드를 추가 하면서 전체 경로의 단축 가능성 확인 및 재설정을 반복
- 효과적인 영역에서 임의로 노드를 선정하는 방식 개선



경로 탐색 (Path Finding) 알고리즘

◆ 경로 탐색 (Path Finding) 알고리즘

Path Finding Algorithm	Features
A*	▪ search (탐색) 기반 알고리즘 ▪ 목표지점까지의 최단 거리 경로를 탐색 ▪ 모든 가능 경로를 탐색하기 때문에 시간이 많이 걸림 ▪ https://en.wikipedia.org/wiki/A*_search_algorithm
RRT	▪ 램덤 샘플링 (random sampling) 기반의 경로 탐색 알고리즘 ▪ Rapidly-exploring Random Tree ▪ 목표 지점까지의 경로를 샘플링 기반으로 탐색하므로 신속하게 탐색 결과를 제공 ▪ 탐색된 결과가 최단 거리 경로임을 보장하지 않음
RRT*	▪ RRT (Rapidly-exploring Random Tree) 알고리즘을 최적화 시킨 것
informed RRT*	▪ https://www.youtube.com/watch?v=nsl-5MZfwu4
PRM	▪ probabilistic roadmap method

A* 경로 탐색 알고리즘

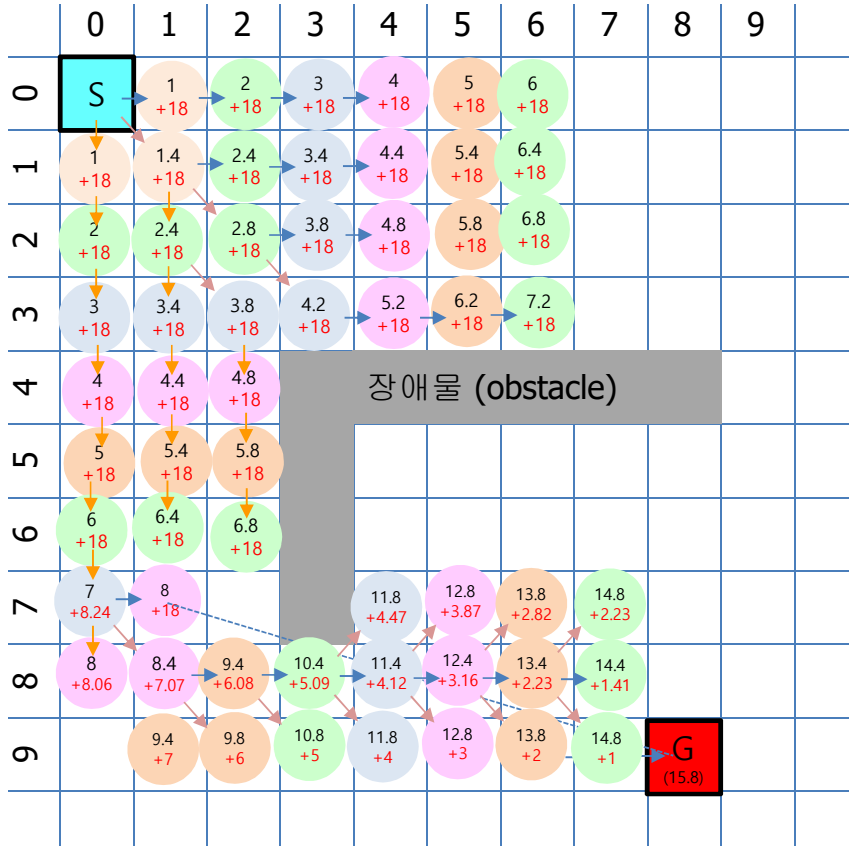
◆ A* search algorithm

- https://en.wikipedia.org/wiki/A*_search_algorithm
- a graph traversal and path search algorithm
- Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968
- It can be seen as an extension of Dijkstra's algorithm
- A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.)
- It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied
- A* achieves better performance by using heuristics to guide its search
- At each iteration of its main loop, A* needs to determine which of its paths to extend
- It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal :
  - f(n) = g(n) + h(n)
  - g(n) : the cost of the path from the start node to *n*
  - h(n) : a heuristic function that estimates the cost of the cheapest path from *n* to the goal.

A* 경로 탐색 알고리즘

◆A* 경로 탐색 알고리즘

- f(n) = g(n) + h(n)
- g(n) : the cost of the path from the start node to *n*
- h(n) : a heuristic function that estimates the cost of the cheapest path from *n* to the goal.
- h(n)의 예:
  - node n와 goal node간의 직선 거리
    h(n) = distance(n, goal)
  - 만약, 이 직선 거리 상에 장애물이 있는 경우,
    h(n) = distance(n, goal)
    = Grid_Width + Grid_Height



A* Algorithm Pseudo Code (1)

```
//  Algorithm A* (Part 1)
function reconstruct_path(cameFrom, current)
        total_path := [current]
        while current in cameFrom.Keys:
                current := cameFrom[current]   // prev[current]
                total_path.append(current)
        return total_path

function A_Star(start, goal, h)
        // function A_Star finds a path from start to goal.
        // h is the heuristic function. h(n) estimates the cost to reach goal from node n.
        // The set of discovered nodes that may need to be (re-)expanded.
        // Initially, only the start node is known.
        // This is usually implemented as a min-heap or priority queue rather than a hash-set.
        openSet := {start}

        // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
        // to n currently known.
        cameFrom := an empty map

        // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
        gScore := map with default value of Infinity
        gScore[start] := 0
```

A* Algorithm Pseudo Code (2)

```
//  A* (Part 2)

    // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
    // how short a path from start to finish can be if it goes through n.
    fScore := map with default value of Infinity
    fScore[start] := h(start)  // heuristic function

    while openSet is not empty
        // This operation can occur in O(1) time if openSet is a min-heap or a priority queue
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, current)
        openSet.Remove(current)
        for each neighbor of current  // neighbor nodes of current in Grid
            // d(current,neighbor) is the weight of the edge from current to neighbor
            // tentative_gScore is the distance from start to the neighbor through current
            tentative_gScore := gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor]
                // This path to neighbor is better than any previous one, do edge relaxation. Record it!
                cameFrom[neighbor] := current  // set prev[]
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)
                if neighbor not in openSet
                    openSet.add(neighbor)
    // Open set is empty but goal was never reached
    return failure
```

A* 경로 탐색 알고리즘의 분석

◆ A* 경로 탐색 알고리즘의 특성

- heuristic 방식으로 목적지에 근접되는 방향의 grid node를 우선적으로 선택하도록 함
- Dijkstra 알고리즘 기반의 경로 탐색에 비교하여 탐색 대상 grid node 수를 줄일 수 있고, 탐색 시간이 단축됨
- heuristic 방식에 따라 최종 선정된 경로의 전체 경로 비용이 달라 질 수 있음

RRT 경로 탐색 알고리즘

◆ RRT (Rapidly-exploring Random Tree)

- https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree
- An RRT grows a tree rooted at the starting configuration by using random samples from the search space.
- As each sample is drawn, a connection is attempted between it and the nearest state in the tree. If the connection is feasible (passes entirely through free space and obeys any constraints), this results in the addition of the new state to the tree. With uniform sampling of the search space, the probability of expanding an existing state is proportional to the size of its Voronoi region. As the largest Voronoi regions belong to the states on the frontier of the search, this means that the tree preferentially expands towards large unsearched areas.
- The length of the connection between the tree and a new state is frequently limited by a **growth factor**. If the random sample is further from its nearest state in the tree than this limit allows, a new state at the maximum distance from the tree along the line to the random sample is used instead of the random sample itself. The random samples can then be viewed as controlling the direction of the tree growth while the growth factor determines its rate. This maintains **the space-filling bias** of the RRT while limiting the size of the incremental growth.
- RRT growth can be biased by increasing the probability of sampling states from a specific area. Most practical implementations of RRTs make use of this to guide the search towards the planning problem goals. This is accomplished by introducing a small probability of sampling the goal to the state sampling procedure. The higher this probability, the more greedily the tree grows towards the goal.

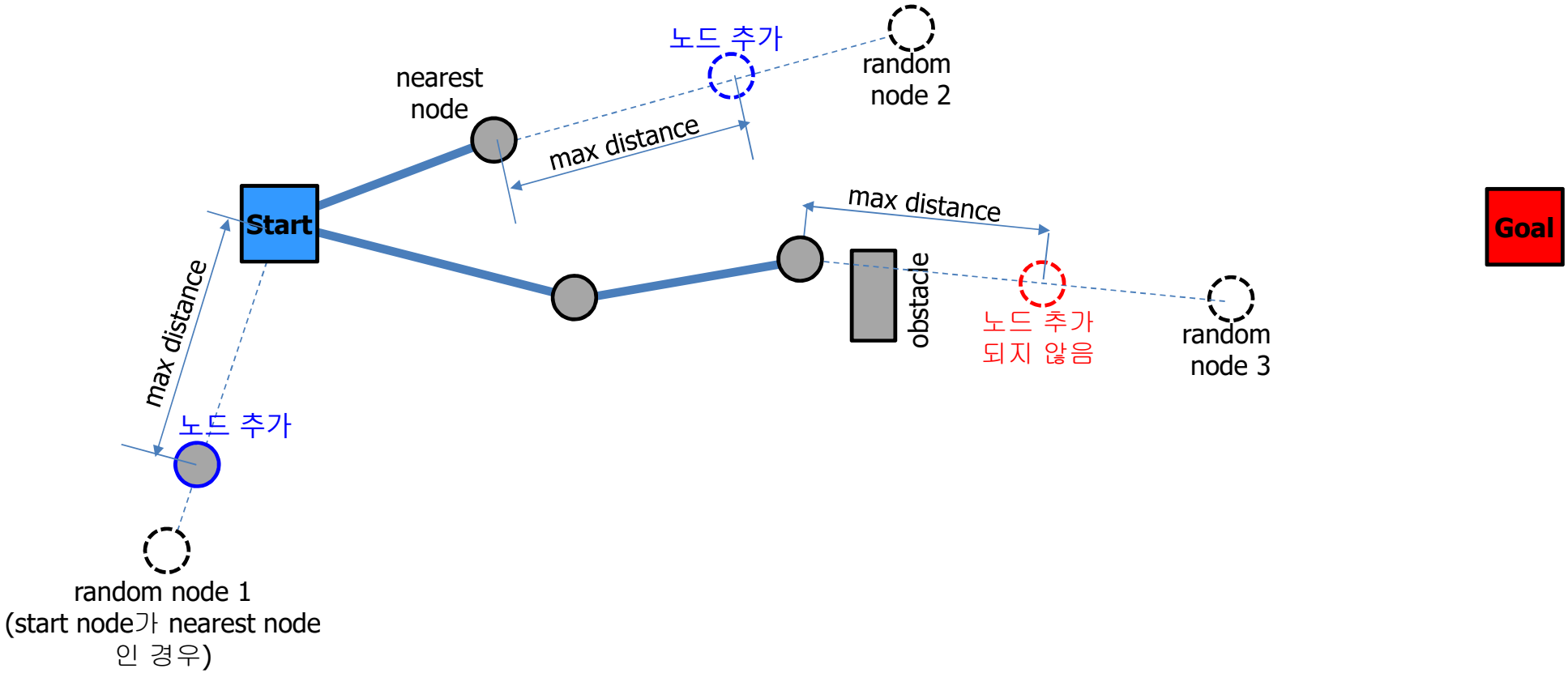
RRT Algorithm

◆ Pseudo Code of RRT Algorithm

```
//Procedure RRT
Qgoal //region that identifies success
Counter = 0 //keeps track of iterations
lim = N //number of iterations algorithm should run for
G(V,E) //Graph containing edges and vertices, initialized as empty
While counter < lim:
    Xnew  = RandomPosition()
    if IsInObstacle(Xnew) == True:
        continue
    Xnearest = Nearest(G(V,E), Xnew) //find nearest vertex
    Link = Chain(Xnew, Xnearest)
    G.append(Link)
    if Xnew in Qgoal:
        Return G
Return G
```

RRT Algorithm

◆ RRT Algorithm의 동작



RRT* 경로 탐색 알고리즘

◆ RRT* 경로 탐색 알고리즘

- RRT (Rapidly-exploring Random Tree) 알고리즘의 최적화

◆ RRT 알고리즘과의 차이점 (1) - cost of each vertex

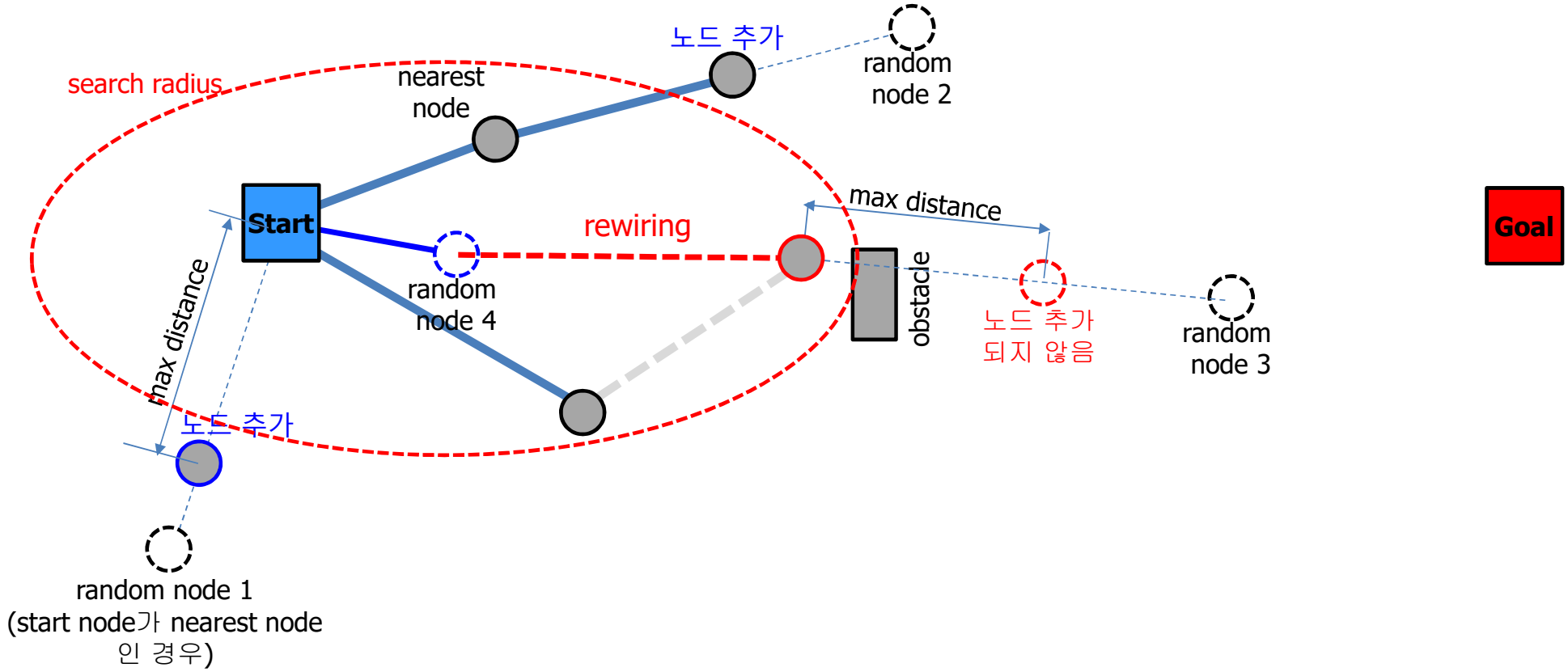
- RRT* records the distance each vertex has traveled relative to its parent vertex; referred as the cost()of the vertex.
- After the closest node is found in the graph, a neighborhood of vertices in a fixed radius from the new node are examined.
- If a node with a cheaper cost() than the proximal node is found, the cheaper node replaces the proximal node.
- The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure.
- The cubic structure of RRT is eliminated.

◆ RRT 알고리즘과의 차이점 (2) - rewiring

- After a vertex has been connected to the cheapest neighbor, the neighbors are again examined.
- Neighbors are checked if being rewired to the newly added vertex will make their cost decrease.
- If the cost does indeed decrease, the neighbor is rewired to the newly added vertex.
- This feature makes the path more smooth.

RRT* Algorithm

◆ RRT* Algorithm의 동작 - rewiring



RRT* 경로 탐색 알고리즘

◆ RRT* Pseudo Code

RRT (Rapidly-exploring Random Tree) Star

```
Rad = r // neighbors 찾기에서 사용되는 반경 (radius)
G(V, E) //Graph containing edges and vertices
For itr in range(0…n)
    Xnew = RandomPosition()
    If Obstacle(Xnew) == True:
        try again
    Xnearest = Nearest(G(V,E), Xnew)
    Cost(Xnew) = Distance(Xnew, Xnearest)
    Xbest, Xneighbors = findNeighbors(G(V,E), Xnew, Rad)
    Link = Chain(Xnew, Xbest)
    For xn in Xneighbors:
        If Cost(Xnew) + Distance(Xnew, xn) < Cost(xn)
            Cost(xn) = Cost(Xnew)+Distance(Xnew, xn)
            Parent(xn) = Xnew // rewiring
            G += {Xnew, xn}
    G += Link   # Link를 Graph G에 추가
Return G
```

A* 알고리즘과 RRT* 알고리즘의 비교

◆ A* 알고리즘과 RRT* 알고리즘의 비교

- A* algorithm is a well-known method in motion planning problems which can find the optimum path between two point in a finite time.
- In contrast the RRT family algorithm, by random sampling from the environment, converges to a collision-free path.
- By simulating these algorithms in complex environments by using java language, it is concluded that RRT family algorithms are significantly faster than A* algorithm
- however the paths which are found by RRT algorithms are longer than A*.

PathFinding 모듈, 자율 주행 제어

pathfinding 모듈

◆ pathfinding 모듈

- https://github.com/brean/python-pathfinding
- https://www.youtube.com/watch?v=8SigT_jhz4I

◆ pathfinding 모듈 설치

```
C:₩Users₩Owner>python -m pip install --upgrade pip
Requirement already satisfied: pip in c:₩users₩owner₩appdata₩local₩programs₩python₩python39₩lib₩site-packages (22.0.2)
Collecting pip
  Downloading pip-22.0.3-py3-none-any.whl (2.1 MB)
     ---------------------------------------- 2.1/2.1 MB 10.2 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 22.0.2
    Uninstalling pip-22.0.2:
      Successfully uninstalled pip-22.0.2
Successfully installed pip-22.0.3

C:₩Users₩Owner>pip install pathfinding
Collecting pathfinding
  Downloading pathfinding-1.0.1-py3-none-any.whl (19 kB)
Installing collected packages: pathfinding
Successfully installed pathfinding-1.0.1
```

pathfinding 모듈을 사용한 예제 (1)

```
# PathFinding with pathfinding module

from pathfinding.core.grid import Grid
from pathfinding.finder.a_star import AStarFinder
from pathfinding.core.diagonal_movement import DiagonalMovement

floor_map = [
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1]]

# create a grid, start and end node
grid = Grid(matrix = floor_map)
start_x, start_y = 0, 0
end_x, end_y = 9, 9
start = grid.node(start_x, start_y)
end = grid.node(end_x, end_y) # (col, row)

finder = AStarFinder(diagonal_movement = DiagonalMovement.always)
path, runs = finder.find_path(start, end, grid)

print("start = ({}, {}), end = ({}, {})".format(start_x, start_y, end_x, end_y))
print("path = ", path)
print("runs = ", runs)
```

```
start = (0, 0), end = (9, 9)
path =  [(0, 0), (1, 1), (2, 2), (2, 3), (2, 4), (2, 5),
(2, 6), (3, 7), (4, 6), (5, 5), (5, 4), (5, 3), (6, 2),
(7, 3), (8, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]
runs =  61
```

```
floor_map = [
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1]]
```

pathfinding 모듈을 사용한 예제 (2)

```
# Path Finding with PathFinding Module on given Floor Map (1)

import pygame, Colors, time, sys
from pathfinding.core.grid import Grid
from pathfinding.finder.a_star import AStarFinder
from pathfinding.core.diagonal_movement import DiagonalMovement

class PathFinder():
    def __init__(self, floor_map_fname, grid_size):
        self.floor_map = pygame.image.load(floor_map_fname)
        self.floor_map_width = self.floor_map.get_width()
        self.floor_map_height = self.floor_map.get_height()
        self.gwin = pygame.display.set_mode((self.floor_map_width, self.floor_map_height))
        print("Floor_map size = {} x {}".format(self.floor_map_width, self.floor_map_height))
            # prepare grid from floor_map
        self.grid_size = grid_size
        self.floor_grid = self.init_floor_grid()
        self.grid = Grid(matrix = self.floor_grid)
        self.finder = AStarFinder(diagonal_movement = DiagonalMovement.always)
        #self.finder = AStarFinder()
```

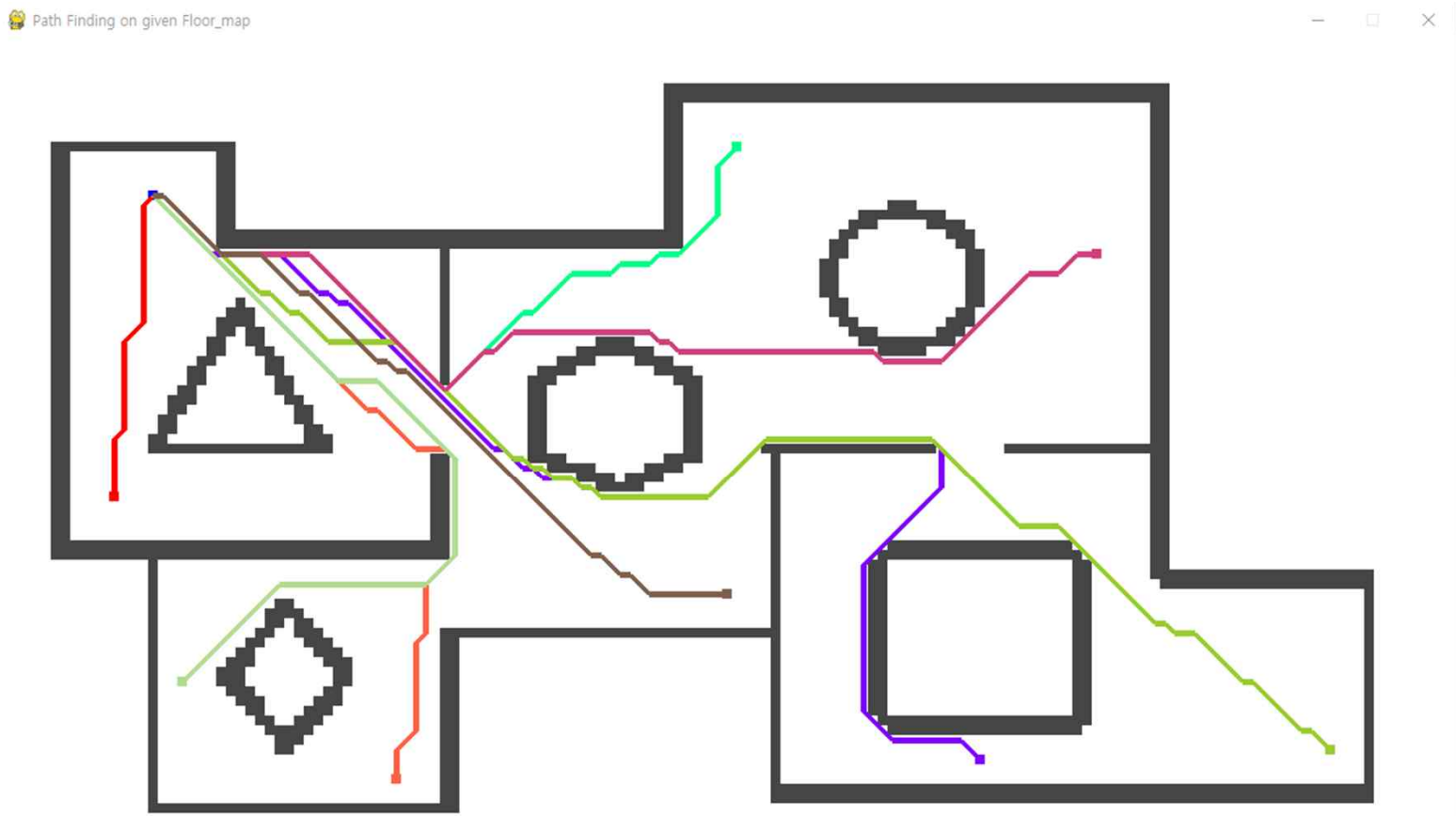
```
# Path Finding with PathFinding Module on given Floor Map (2)

    def init_floor_grid(self):
        floor_grid = []
        grid_size = self.grid_size
        floor_grid_width, floor_grid_height = self.floor_map_width//grid_size, self.floor_map_height//grid_size
        #print("floor_map_width = {} => floor_grid_width = {}".format(self.floor_map_width, floor_grid_width))
        #print("floor_map_height = {} => floor_grid_height = {}".format(self.floor_map_height, floor_grid_height))
        for r in range(floor_grid_height+1):
            floor_grid_row = []
            for c in range(floor_grid_width+1):
                floor_grid_row.append(1) # initial value 1 as free space
            floor_grid.append(floor_grid_row)
        print("len(floor_grid) = {}, len(floor_grid[0]) = {}".format(len(floor_grid), len(floor_grid[0])))
        #print("floor_grid = \n", floor_grid)
        for r in range(self.floor_map_height):
            for c in range(self.floor_map_width):
                spot_color = self.floor_map.get_at((c, r))
                if spot_color == Colors.Black:
                    gr, gc = r//self.grid_size, c//grid_size
                    floor_grid[gr][gc] = 0 # 0 means obstacle

        # print floor_grid
        print("floor_grid ({} x {}) ".format(len(floor_grid), len(floor_grid[0])))
        """
        for r in range(len(floor_grid)):
            for c in range(len(floor_grid[0])):
                print("{:2d}".format(floor_grid[r][c]), end=' ')
            print()
        """
        return floor_grid
```

```
# Path Finding with PathFinding Module on given Floor Map (3)

    def show_floor_grid(self):
        grid_size = self.grid_size
        for r in range(len(self.floor_grid)):
            for c in range(len(self.floor_grid[0])):
                x = c * grid_size - grid_size //2
                y = r * grid_size - grid_size //2
                grid_block = (x, y, grid_size, grid_size)
                if self.floor_grid[r][c] == 0:
                    pygame.draw.rect(self.gwin, Colors.Grey, grid_block)
                else:
                    pygame.draw.rect(self.gwin, Colors.White, grid_block)
        pygame.display.update()

    def find_path(self, start_gpos, end_gpos):
        self.floor_grid = self.init_floor_grid()
        self.grid = Grid(matrix = self.floor_grid)
        self.finder = AStarFinder(diagonal_movement = DiagonalMovement.always)
        start_gx, start_gy = start_gpos # (col, row)
        end_gx, end_gy = end_gpos # (col, row)
        start_node = self.grid.node(start_gx, start_gy)
        end_node = self.grid.node(end_gx, end_gy) # (col, row)
        path, runs = self.finder.find_path(start_node, end_node, self.grid)
        # show path on gwin
        if len(path) != 0:
            prev_pos = (path[0][0] * grid_size, path[0][1] * grid_size)
            for gpos in path:
                pos = (gpos[0] * grid_size, gpos[1] * grid_size)
                pygame.draw.line(self.gwin, path_colors[path_count % len(path_colors)], prev_pos, pos, 5)
                prev_pos = pos
        return path, runs
```

```
# Path Finding with PathFinding Module on given Floor Map (4)

# ------------------------------
path_colors = Colors.Color_List
floor_map_fname = "floor_map0.png"
grid_size = 8
if __name__ == '__main__':
    pygame.init()
    PF = PathFinder(floor_map_fname, grid_size)
    pygame.display.set_caption("Path Finding on given Floor_map")
    pygame.mouse.set_cursor(*pygame.cursors.diamond)

    start_gx, start_gy = 16, 16 # coordinates on grid
    start_grid_pos = (start_gx, start_gy)
    end_gx, end_gy = 20, 5
    mark_size = grid_size

    # show floor_grid
    PF.show_floor_grid()
    start_x, start_y = start_gx * grid_size - grid_size //2, start_gy * grid_size - grid_size //2
    start_mark = (start_x, start_y, mark_size, mark_size) # pos_x, pos_y, width, height
    pygame.draw.rect(PF.gwin, Colors.Blue, start_mark)

    update_path_flag = False
    path_count = 0
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
```

```
# Path Finding with PathFinding Module on given Floor Map (4)

            if pygame.mouse.get_focused():
                mouse_pos = pygame.mouse.get_pos() # returns (x, y)
                mouse_gx, mouse_gy = mouse_pos[0] // grid_size, mouse_pos[1] // grid_size
                if PF.floor_grid[mouse_gy][mouse_gx] == 1 and pygame.mouse.get_pressed() == (1, 0, 0):
                    end_gx, end_gy = mouse_pos[0] // grid_size, mouse_pos[1] // grid_size
                    end_grid_pos = (end_gx, end_gy )

                    print("{}-th end grid_pos = ({})".format(path_count, (end_gx, end_gy)))
                    end_x, end_y = end_gx * grid_size - grid_size //2, end_gy * grid_size - grid_size //2
                    end_mark = (end_x, end_y, mark_size, mark_size) # pos_x, pos_y, width, height
                    pygame.draw.rect(PF.gwin, path_colors[path_count % len(path_colors)], end_mark)
                    pygame.display.update()
                    update_path_flag = True

                    path, runs = PF.find_path(start_grid_pos, end_grid_pos)
                    print("start_grid_pos = ({}), {}-th end_grid_pos = ({})".format(start_grid_pos, path_count, end_grid_pos))
                    print("path = ", path)
                    print("runs = ", runs)
                    if len(path) == 0:
                        print("Path not found !!")
                        continue
                    path_count += 1

                else:
                    continue
        pygame.display.update()
        if update_path_flag == False:
            time.sleep(0.1)
            continue
```

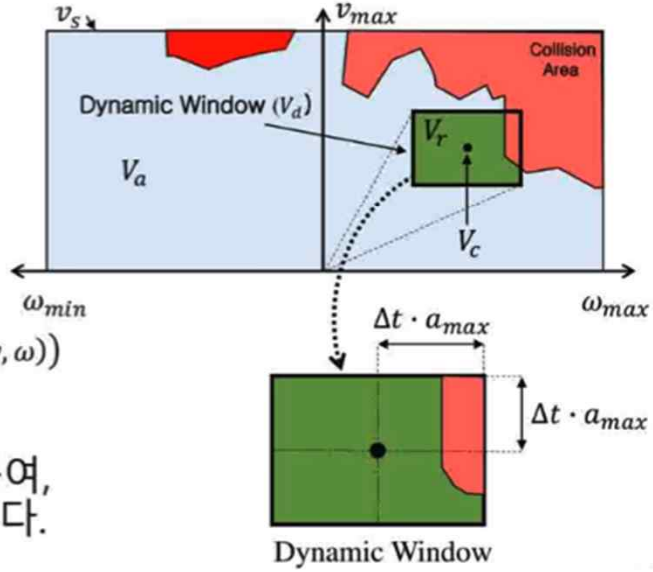
pathfinding 모듈을 사용한 예제 (2) - 실행 결과



Navigation with Dynamic Window Approach (DWA)

◆ Navigation – Dynamic Window Approach (DWA)

- **Dynamic Window Approach** (local plan에서 주로 사용)
- 로봇의 속도 탐색 영역(velocity search space)에서 로봇과 충돌 가능한 장애물을 회피하면서 목표점까지 빠르게 다다를 수 있는 속도를 선택하는 방법

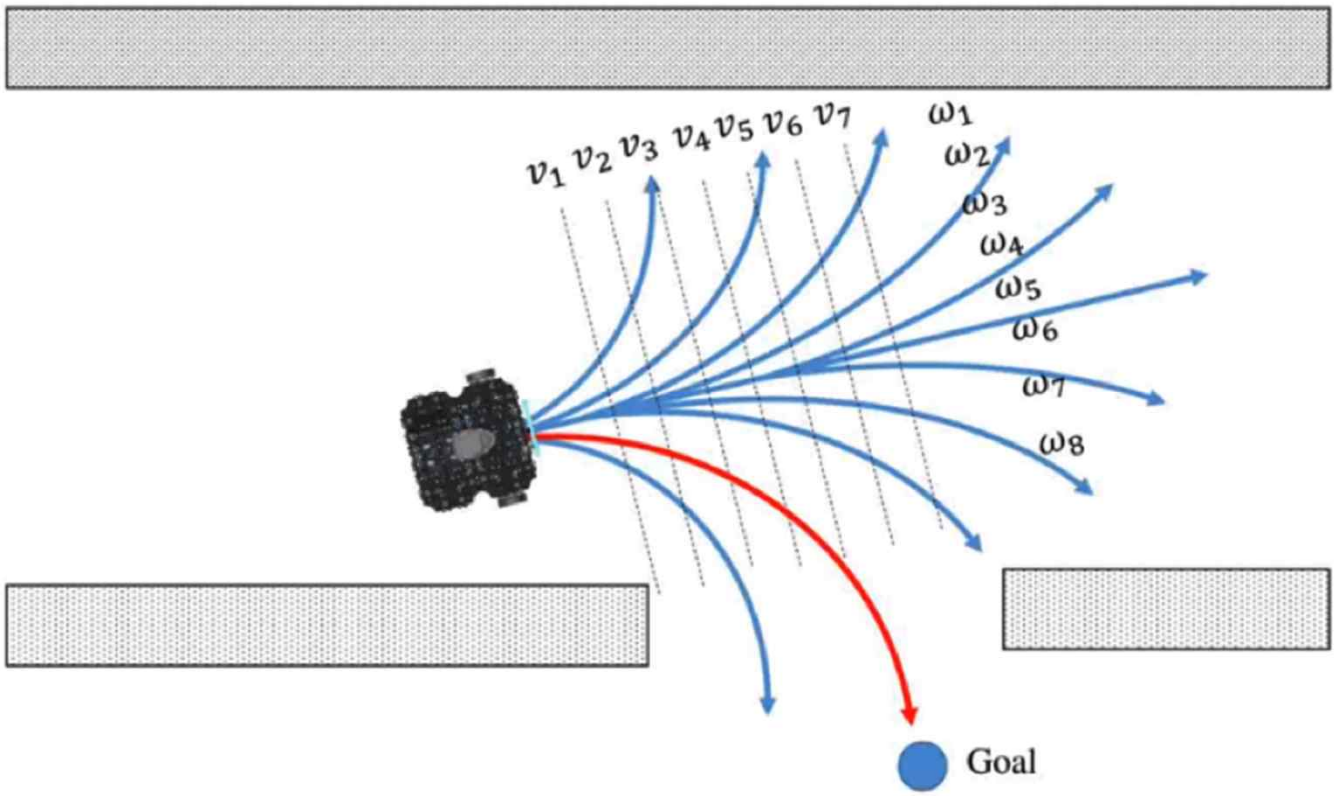


- v (병진속도), ω (회전속도)
- V_s: 가능 속도 영역
- V_a: 허용 속도 영역
- V_r: 다이나믹 윈도우 안의 속도 영역
- $G_{(v,\omega)} = \sigma(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega))$

- 목적함수 G는 로봇의 방향, 속도, 충돌을 고려하여, 목적함수가 최대가 되는 속도 v, ω 를 구하게 된다.

Dynamic Window Approach (DWA)

◆ v(병진속도)와 w(회전속도)에 따른 궤적 예측

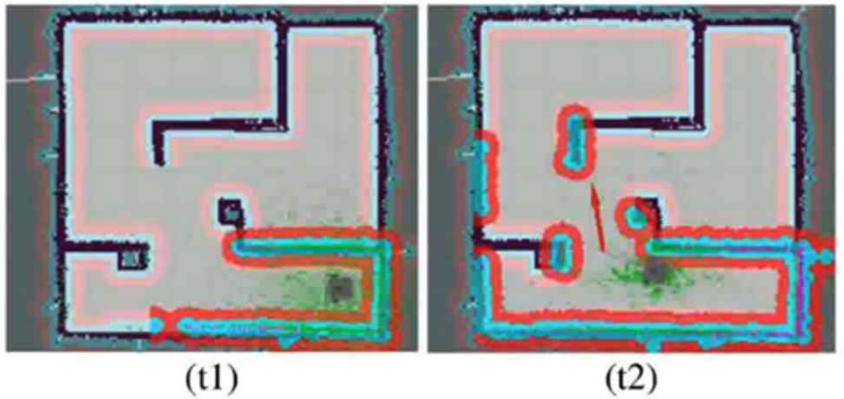


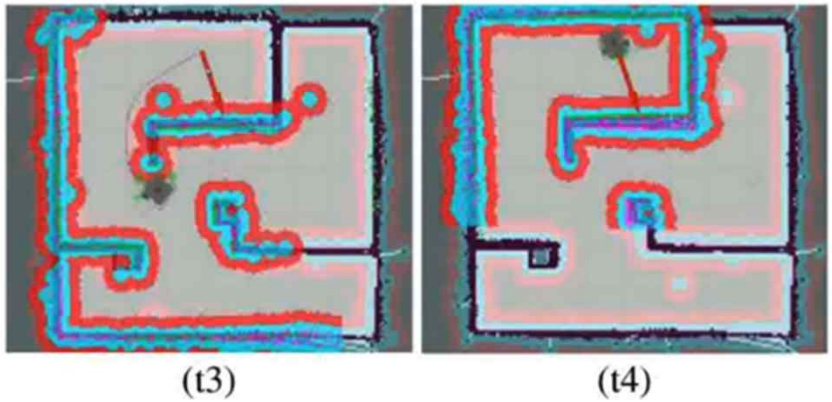
이동 중의 위치 추정 (localization)

위치 추정(localization) | Kalman filter, Particle filter, Graph, Bundle adjustment

- 파티클 필터(Particle Filter)
- 파티클 필터는 시행 착오(try-and-error)법을 기반으로한 시뮬레이션을 통하여 예측하는 기술으로 대상 시스템에 확률 분포로 임의로 생성된 추정값을 파티클(입자) 형태로 나타낸다.

1) 초기화(initialization)
2) 예측(prediction)
3) 보정(update)
4) 위치 추정(pose estimation)
5) 재추출(Resampling)





차선 유지 및 주행 속도 제어

◆ 자율 주행에서 차선을 유지

- 영상 정보를 기반으로 차선 유지를 확인

◆ 자율 주행 중 주행 속도의 지능적 제어

- 코너링에서 주행속도가 높은 경우 차체가 원심력에 의하여 기울어지므로, 속도를 감속
- 직선 구간의 경우, 최대 허용 속도로 주행
- 도로 표지판에 따른 속도 제어

참고문헌

[1] https://docs.px4.io/master/ko/.

[2] Sebastian Thrun, Wolfram Burgard and Dieter Fox, Probabilistic Robotics, MIT press, August 2005.

[3] 시각 관성 주행거리 측정(VIO), https://docs.px4.io/master/ko/computer_vision/visual_inertial_odometry.html.

[4] Pixhawk와 ROS를 이용한 자율주행 드론, https://dnddnjs.gitbooks.io/drone-autonomous-flight/content/.

[5] Drone Programming With Python Course | 3 Hours | Including x4 Projects | Computer Vision, https://www.youtube.com/watch?v=LmEcyQnfpDA.

[6] https://google-cartographer.readthedocs.io/en/latest/

[7] https://ichi.pro/ko/google-cartographer-mich-rplidarwa-raspberry-pileul-sayonghan-2d-maeping-186147659465925

[8] https://github.com/cartographer-project/cartographer_ros

[9] https://elecs.tistory.com/296

[10] https://linklab-uva.github.io/autonomousracing/assets/files/SLAM.pdf

[11] 오로카 cartographer 강연 20201201 – YouTube

[12] https://google-cartographer-ros.readthedocs.io/en/latest/assets_writer.html

[13] 강화학습을 이용한 자율주행 구현, https://github.com/NOHYC/autonomous_driving_car_project.

[14] Mini FSESC4.20 50A base on VESC® 4.12 with Aluminum Anodized Heat Sink, https://flipsky.net/products/mini-fsesc4-20-50a-base-on-vesc-widely-used-in-eskateboard-escooter-ebike.

[15] 미니 FSESC4.20 50A 베이스, VESC®알루미늄 알루마이트 방열판 Flipsky 4.12, Ali Express, KRW 118,211, https://ko.aliexpress.com/item/4000438827676.html?gatewayAdapt=glo2kor&spm=a2g0o.search0302.0.0.4c2fb920WxB9Ld&algo_pvid=4f45efd9-8eb9-41f3-ac5d-29d1bd1c3155&algo_exp_id=4f45efd9-8eb9-41f3-ac5d-29d1bd1c3155-4

참고문헌

[16] Get Your Motor Running! - VESC - Jetson RACECAR, https://www.youtube.com/watch?v=fiaiA-o83c4
[17] JetsonHacks, Jetson Race Car/J, https://www.jetsonhacks.com/racecar-j/.
[18] Understanding A* Path Algorithms and Implementation with Python, https://towardsdatascience.com/understanding-a-path-algorithms-and-implementation-with-python-4d8458d6ccc7.
[19] Python Implementation of A* Algorithm, https://github.com/ademakdogan/Implementation-of-A-Algorithm-Visualization-via-Pyp5js-.
[20] Path Planning with A* and RRT | Autonomous Navigation, Part 4, https://www.youtube.com/watch?v=QR3U1dgc5RE.
[21] RRT STAR | RRT* | Path Planning Algorithm | Python Code, https://www.youtube.com/watch?v=M5Q6Fywd36w.
[22] RRT, RRT* & Random Trees, https://www.youtube.com/watch?v=Ob3BIJkQJEw.
[23] Python implementation of Rapidly-exploring random tree (RRT) path-planning algorithm, https://gist.github.com/Fnjn/58e5eaa27a3dc004c3526ea82a92de80.
[24] The RRT path planning algorithm simulated with python | part 1, https://www.youtube.com/watch?v=TzfNzqjJ2VQ.
[25] The RRT path planning algorithm simulated with python | part 2, https://www.youtube.com/watch?v=JpKkfWxbqqg.
[26] The RRT path planning algorithm simulated with python | part 3, https://www.youtube.com/watch?v=BHG8VKwEPuw.
[27] The RRT path planning algorithm simulated with python | part 4, https://www.youtube.com/watch?v=vAoDnjgIVKU.
[28] RRT* FND - motion planning in dynamic environments, https://www.youtube.com/watch?v=hXTnWN8NiKE.
[29] Easy pathfinding in python [almost without math], https://www.youtube.com/watch?v=8SigT_jhz4I.
[30] Motion Planning Algorithms (RRT, RRT*, PRM) - [MIT 6.881 Final Project], https://www.youtube.com/watch?v=gP6MRe_IHFo.
[31] Dynamic RRT (demo), https://www.youtube.com/watch?v=TA0_0Zb7cjE.