3-3 최단 경로 알고리즘

강의 요약

01

Queue (큐)

FIFO

02

BFS

- Queue
- 시간 복잡도: O(V+E)
- 공간 복잡도: O(V)
- 목표 노드가 시작점 근처에 있을 때
- 가중치가 모두 동일할 경우 최단 경로 탐색 가능

03

Stack (스택)

LIFO

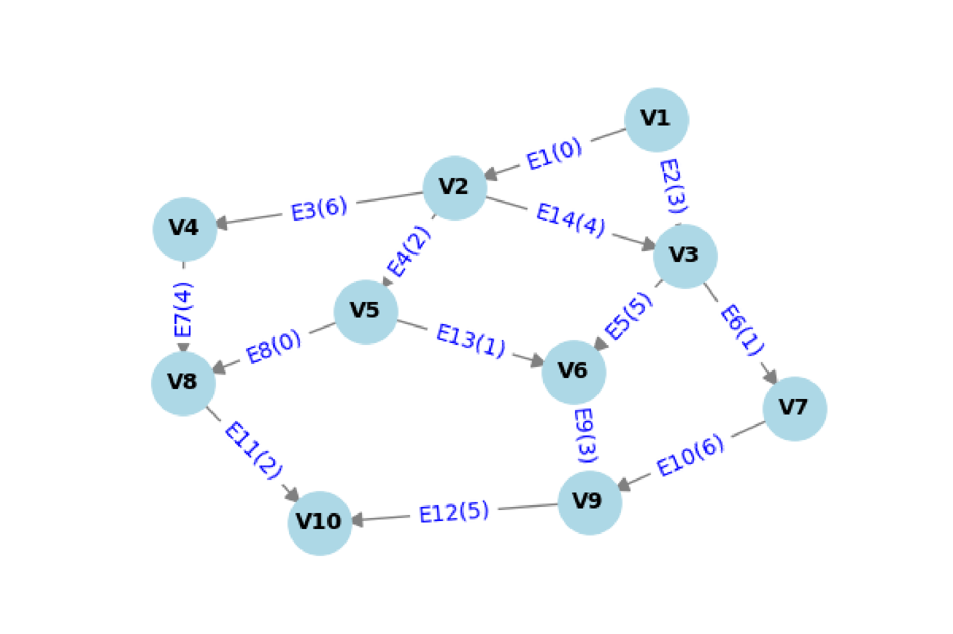
04

DFS

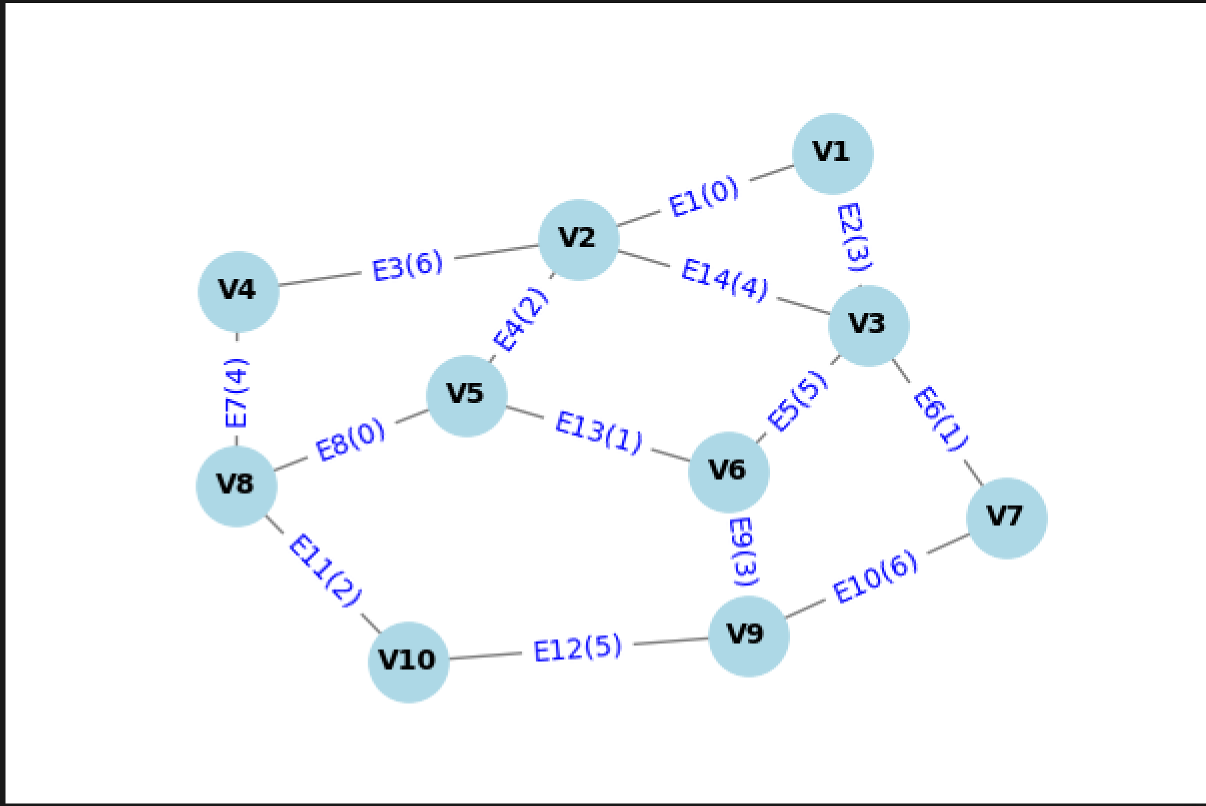
- Stack (LIFO) 사용
- 시간 복잡도: O(V+E)
- 공간 복잡도: O(V) 일반적으로 BFS보다 메모리 사용량이 낮음
- 일반적으로 최단 경로를 보장하지 않음

그래프 탐색 알고리즘의 필요성

테스크 → 데이터 구조로 문제를 정의 (그래프) → 경로 탐색 → 경로

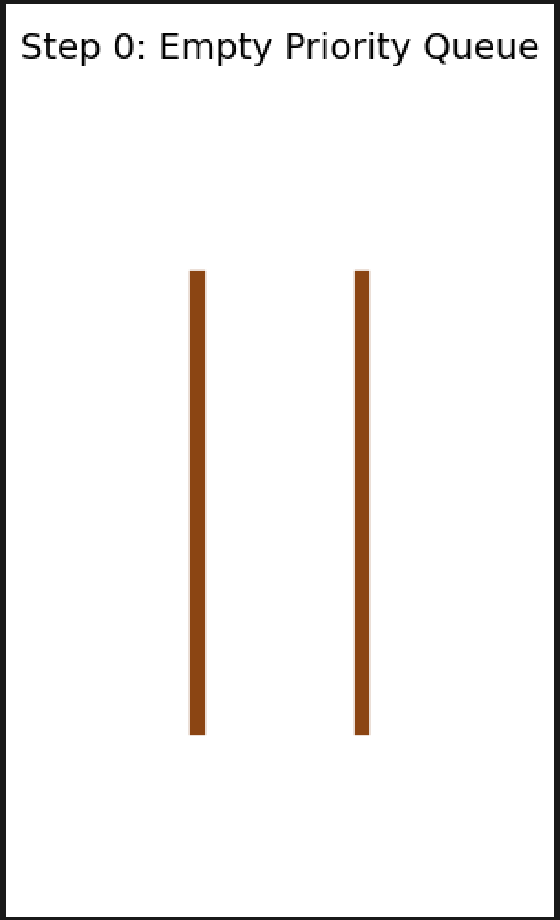


최단 경로 탐색의 기초 설명



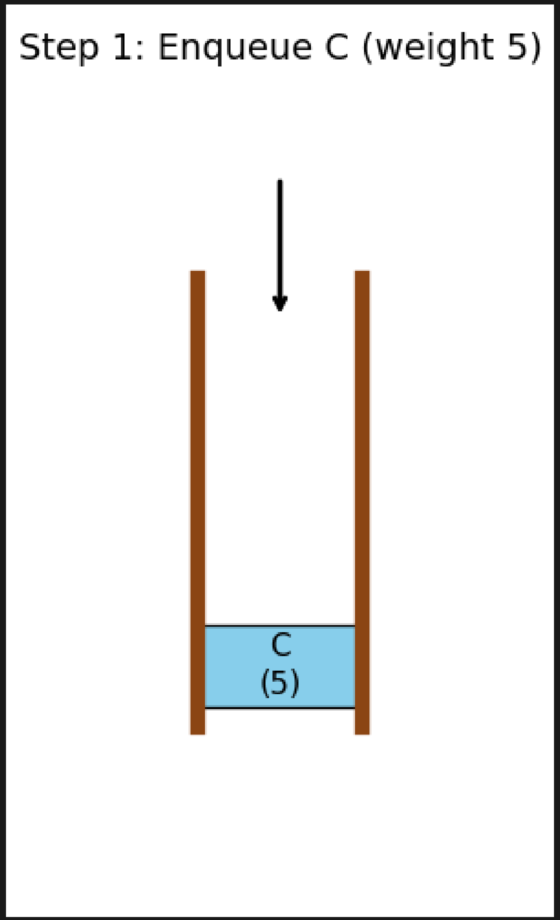
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



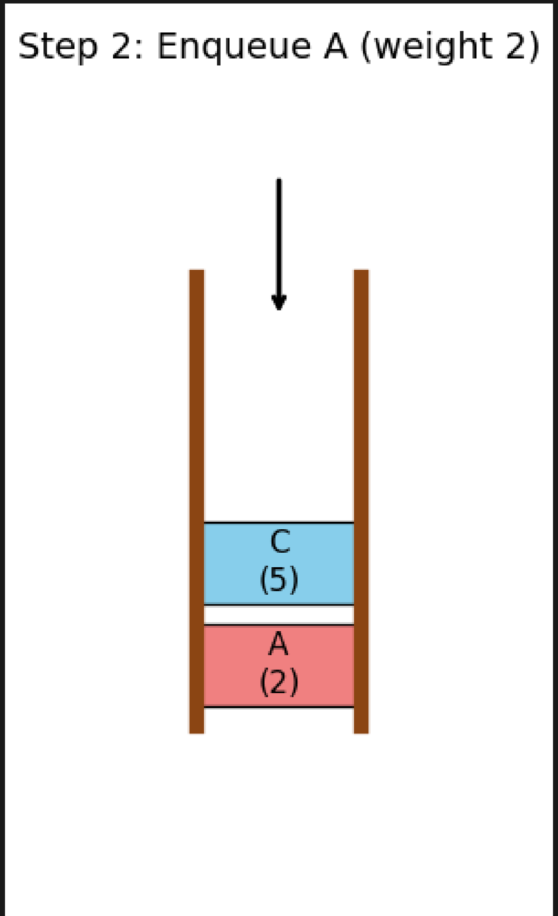
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



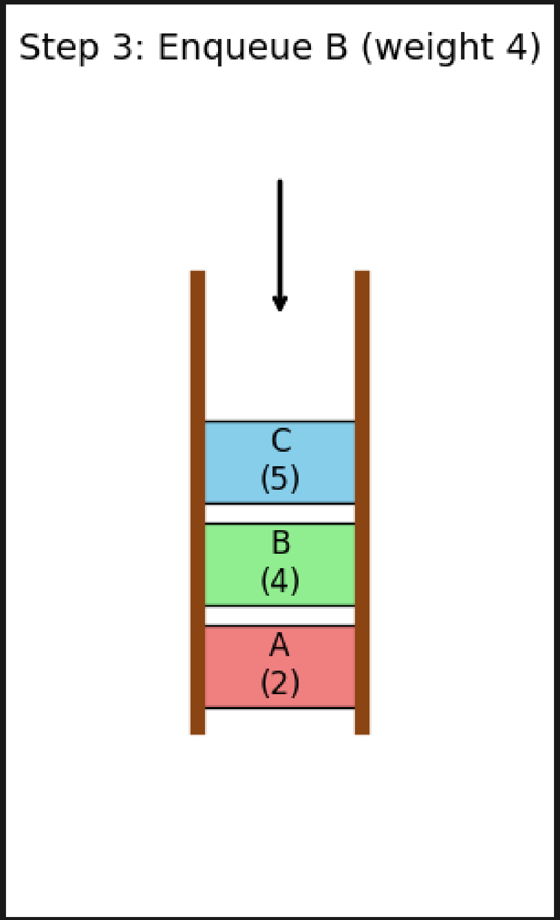
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



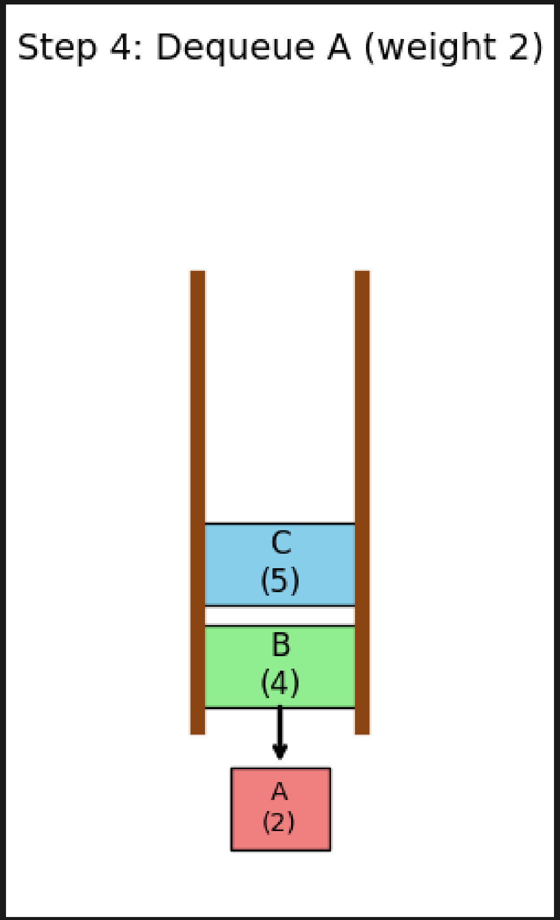
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



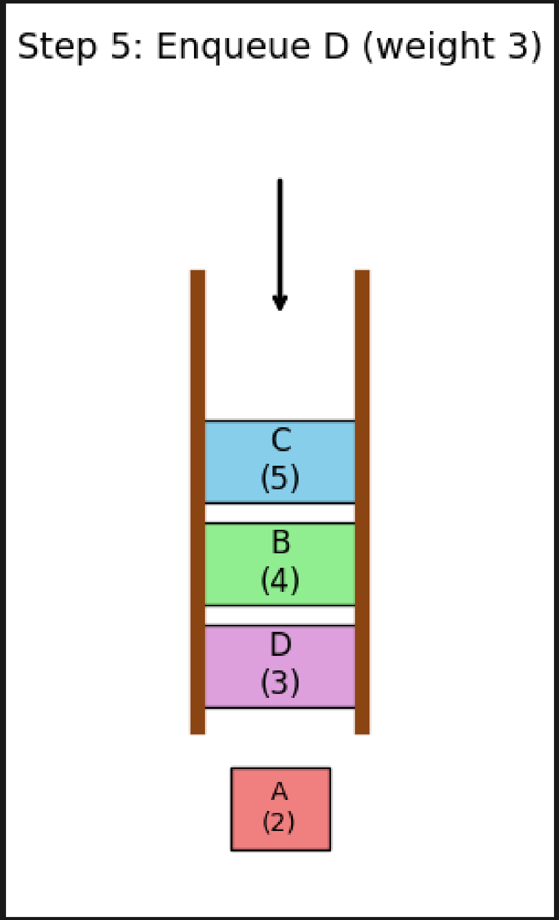
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



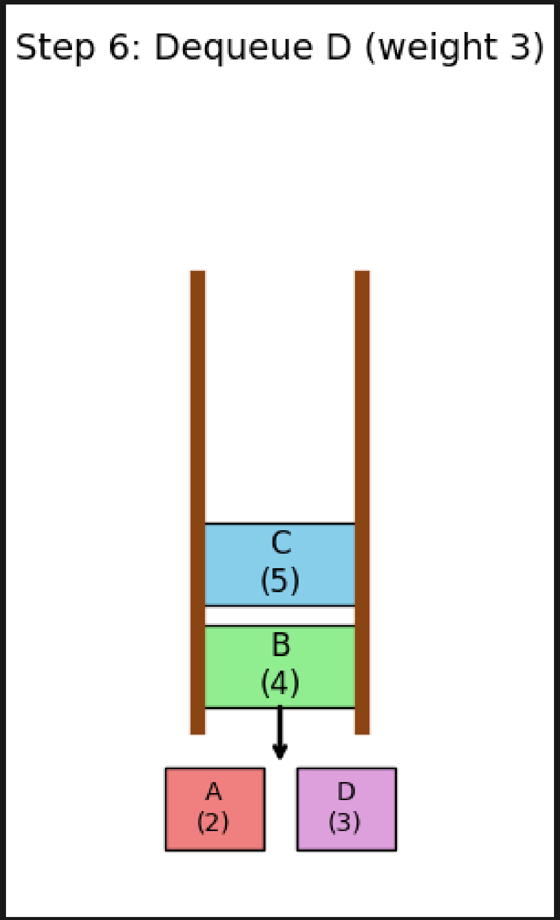
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



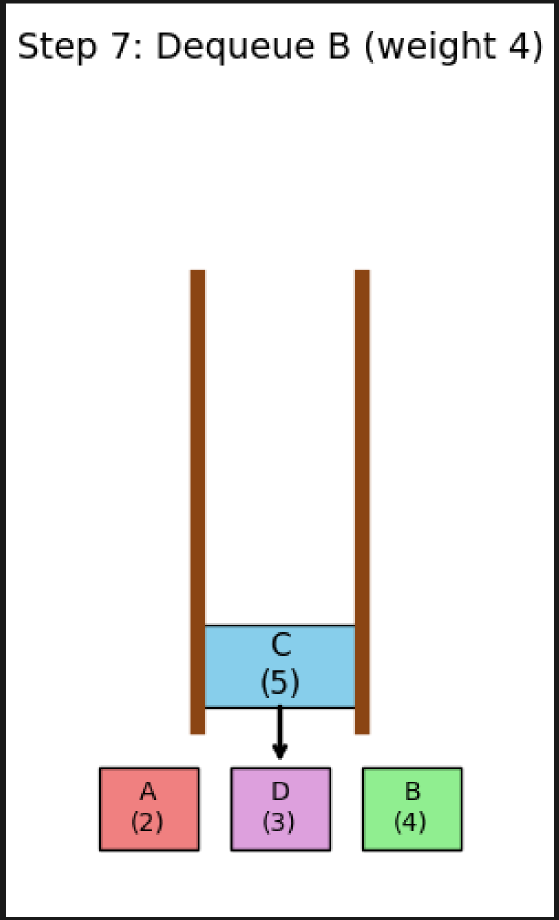
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



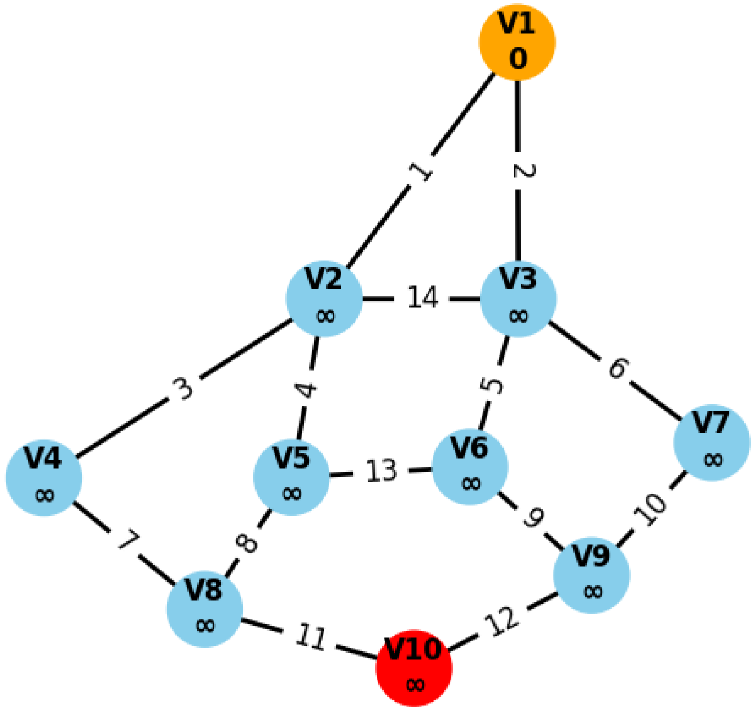
다익스트라 알고리즘 (Dijkstra Algorithm)

Priority Queue: First In First Out (FIFO)



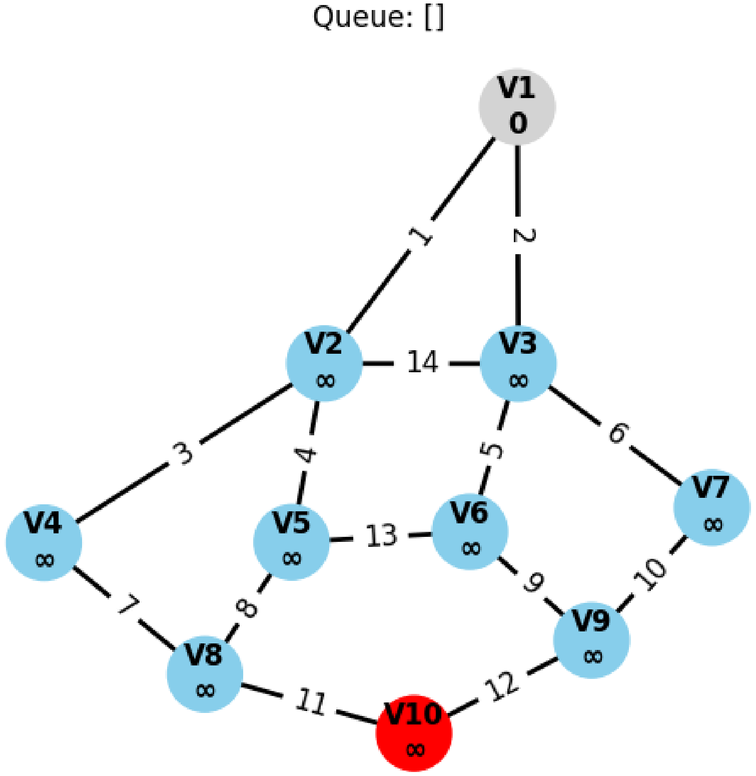
다익스트라 알고리즘 (Dijkstra Algorithm)

Queue: [(0, 'V1')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	∞	
V3	∞	
V4	∞	
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

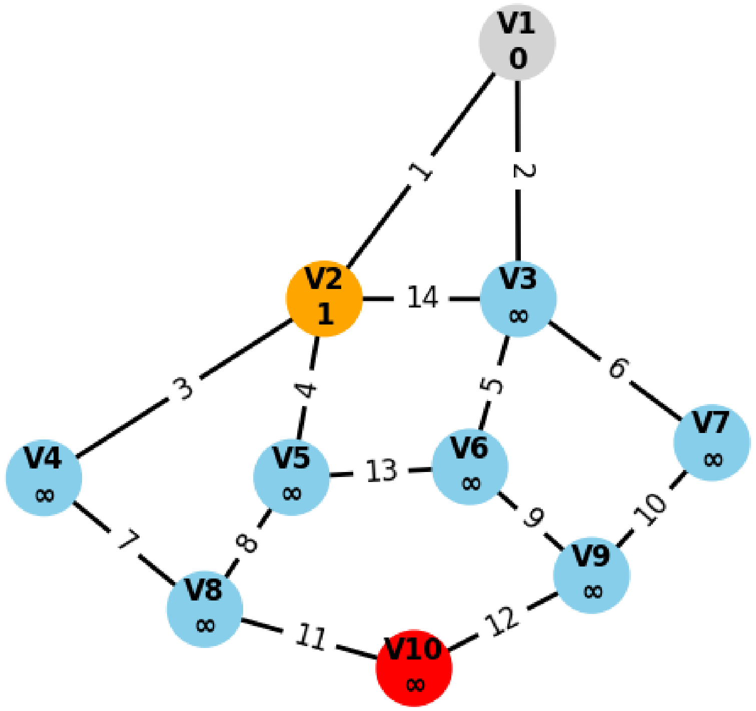
다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	∞	
V3	∞	
V4	∞	
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

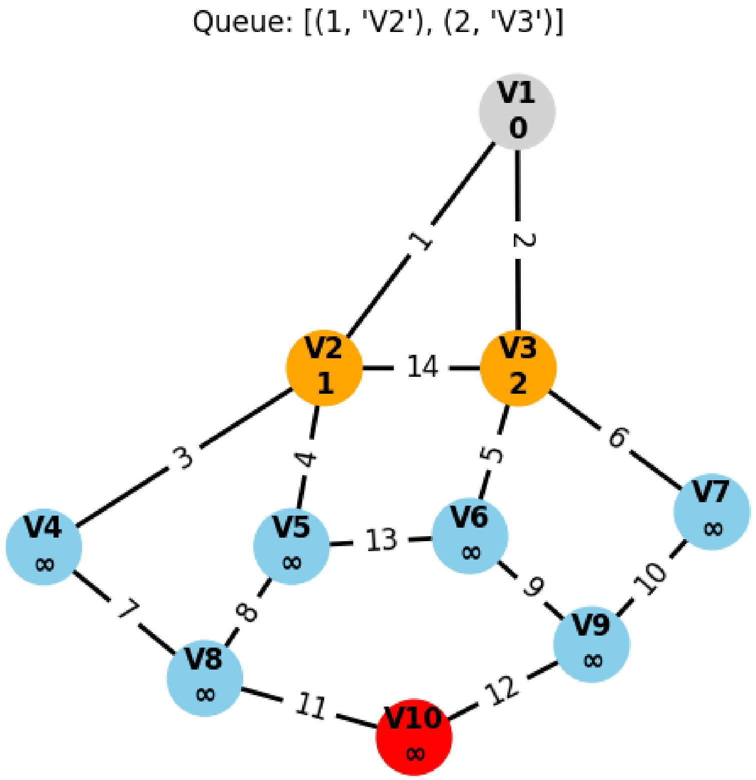
다익스트라 알고리즘 (Dijkstra Algorithm)

Queue: [(1, 'V2')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	∞	
V4	∞	
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

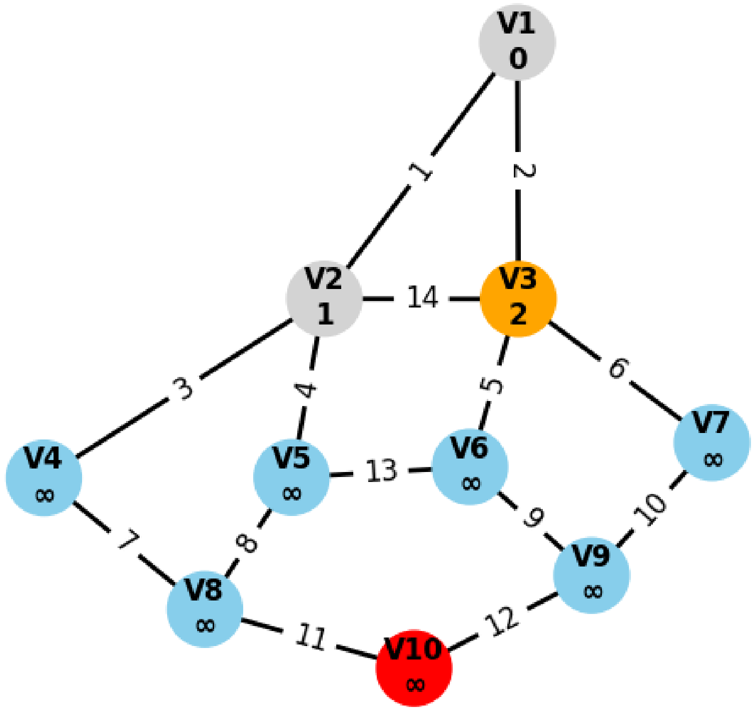
다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	∞	
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

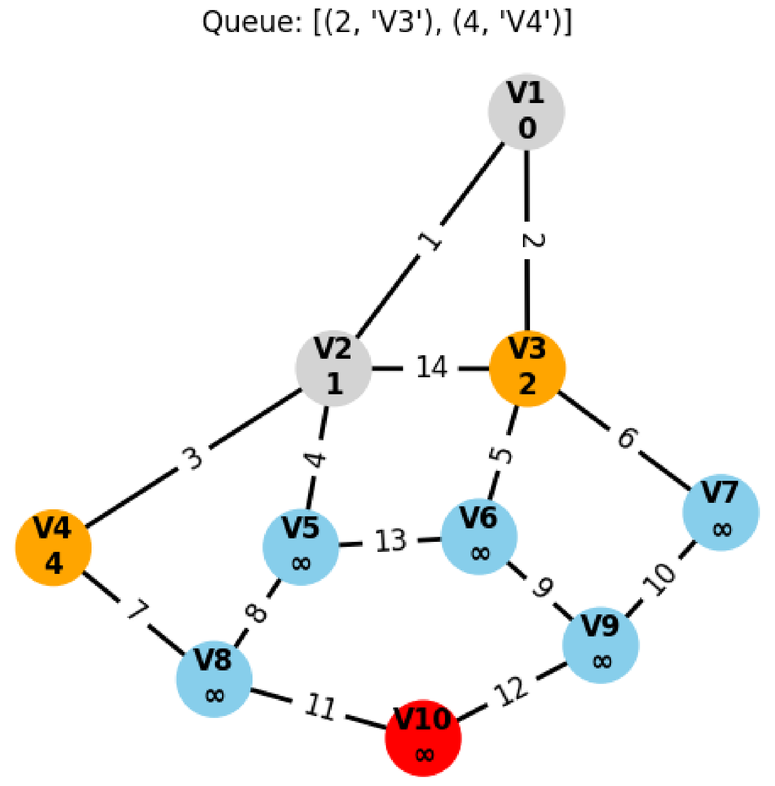
다익스트라 알고리즘 (Dijkstra Algorithm)

Queue: [(2, 'V3')]



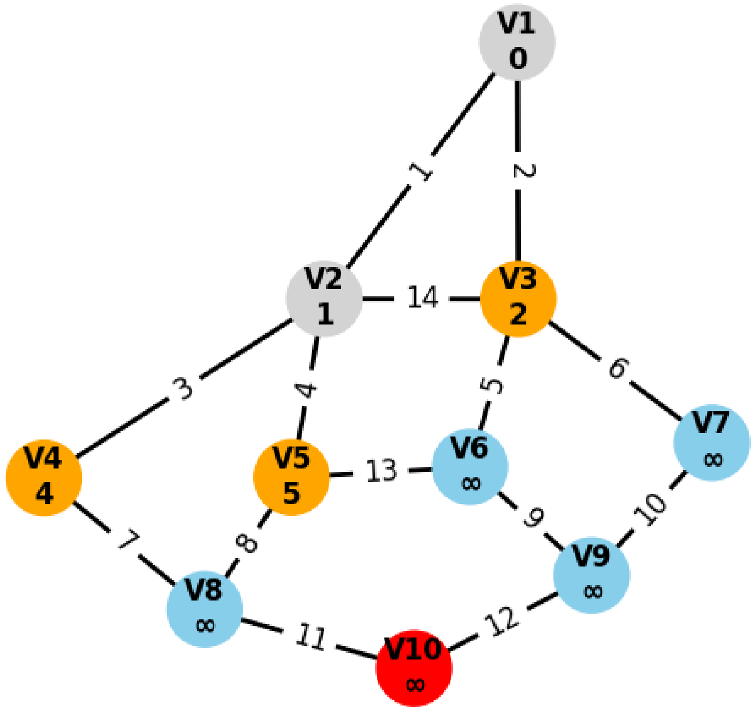
Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	∞	
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)



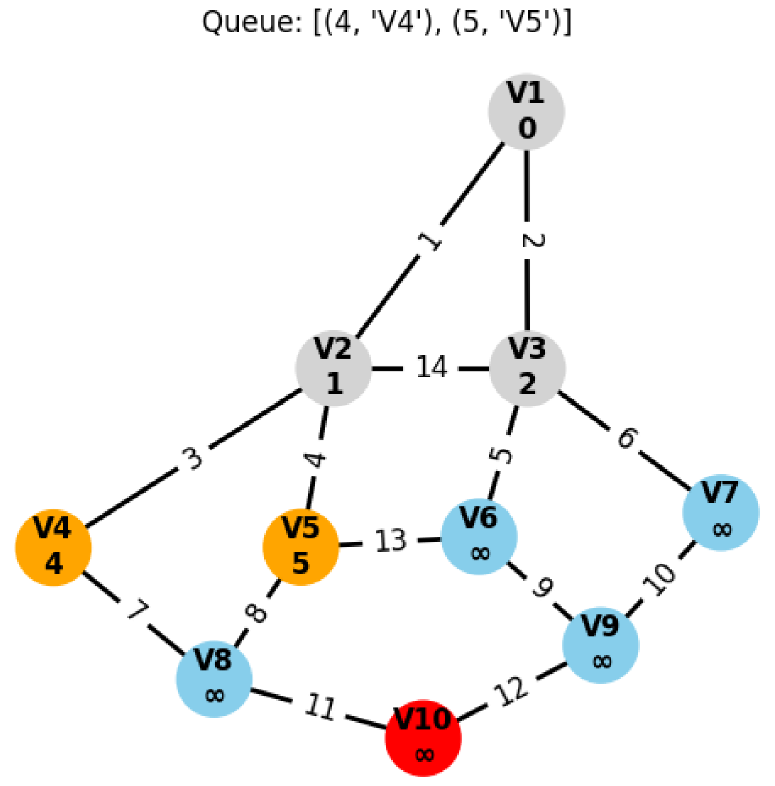
Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	∞	
V6	∞	
V7	∞	
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	∞	
V7	∞	
V8	∞	
V9	∞	

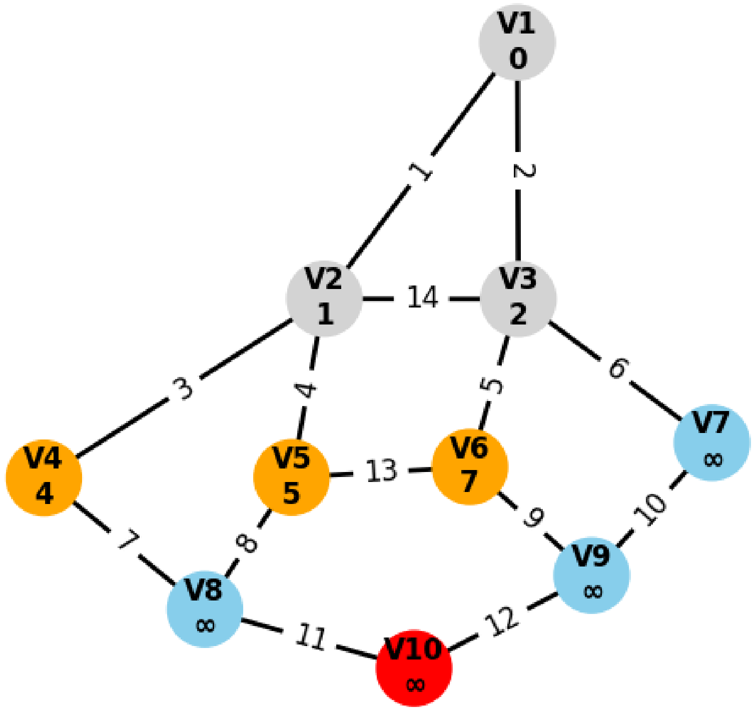
다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	∞	
V7	∞	
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

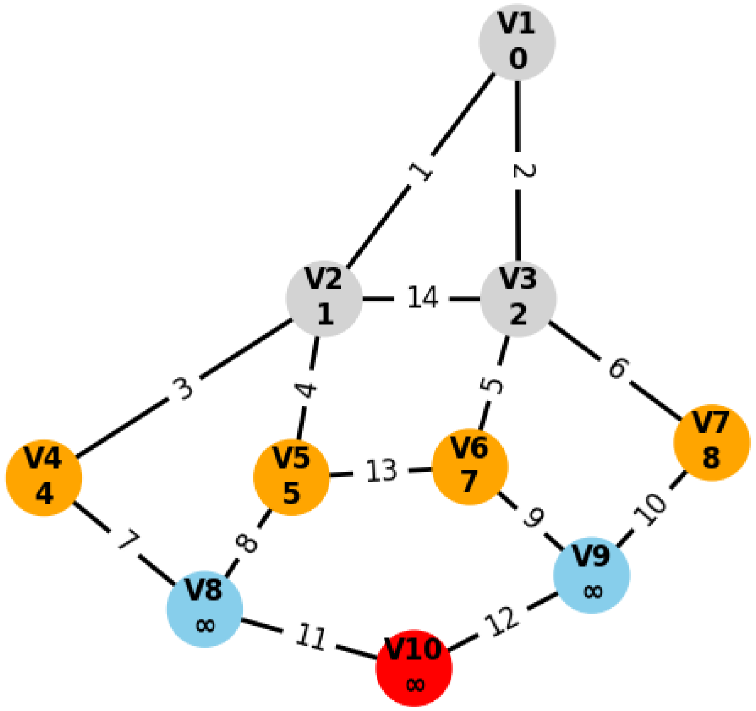
Queue: [(4, 'V4'), (5, 'V5'), (7, 'V6')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	∞	
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

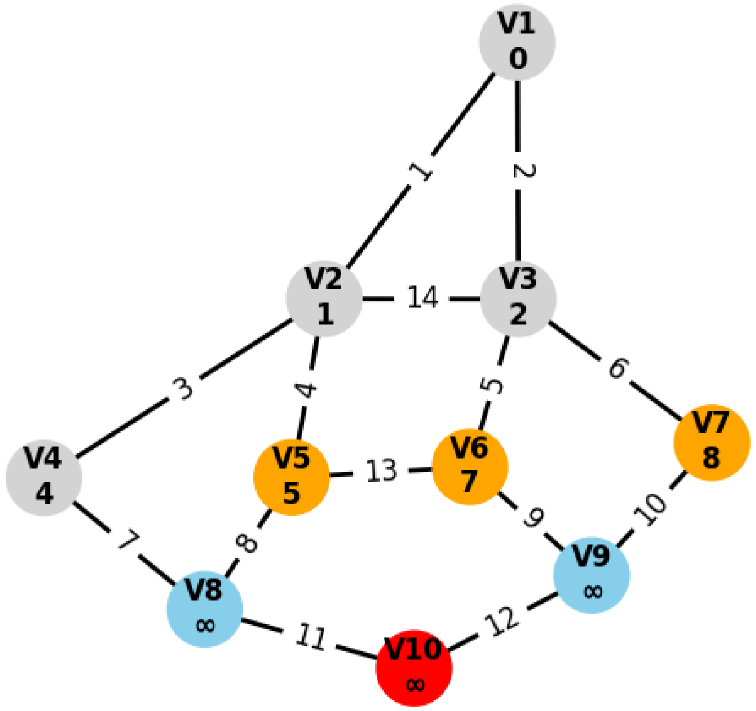
Queue: [(4, 'V4'), (5, 'V5'), (7, 'V6'), (8, 'V7')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

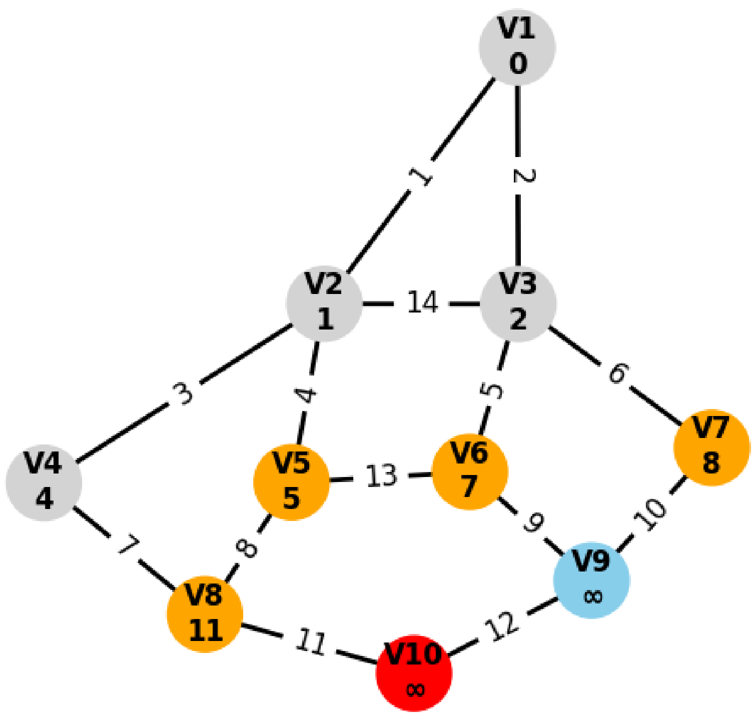
Queue: [(5, 'V5'), (8, 'V7'), (7, 'V6')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	∞	
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

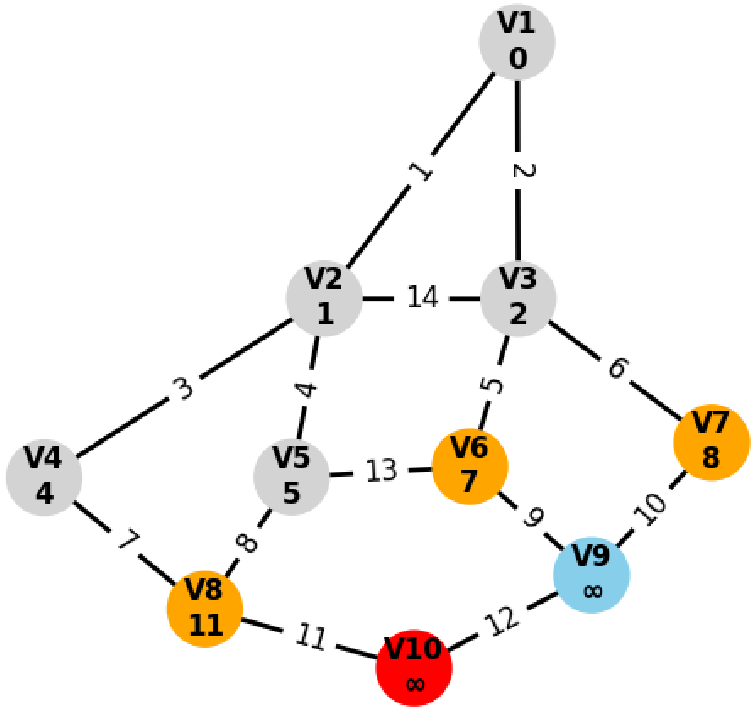
Queue: [(5, 'V5'), (8, 'V7'), (7, 'V6'), (11, 'V8')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

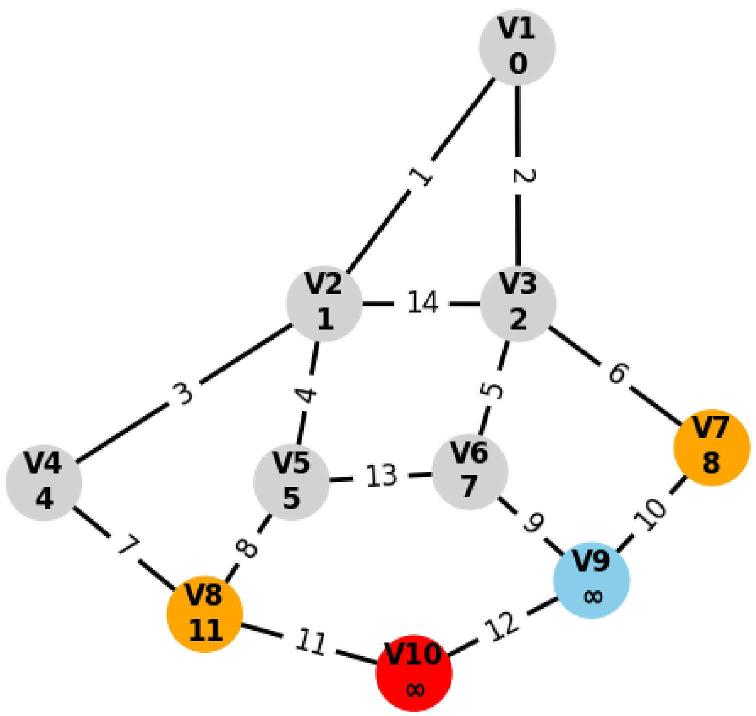
Queue: [(7, 'V6'), (8, 'V7'), (11, 'V8')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

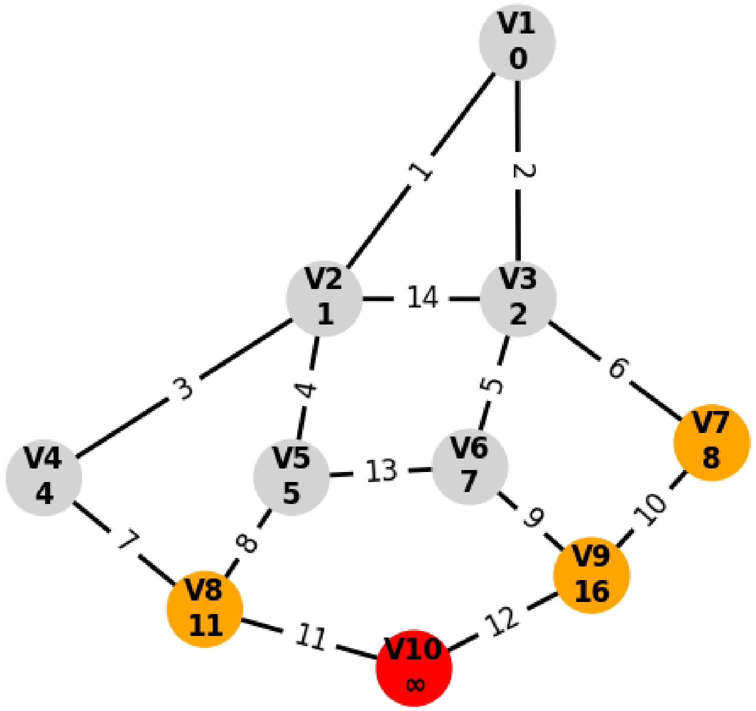
Queue: [(8, 'V7'), (11, 'V8')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	∞	

다익스트라 알고리즘 (Dijkstra Algorithm)

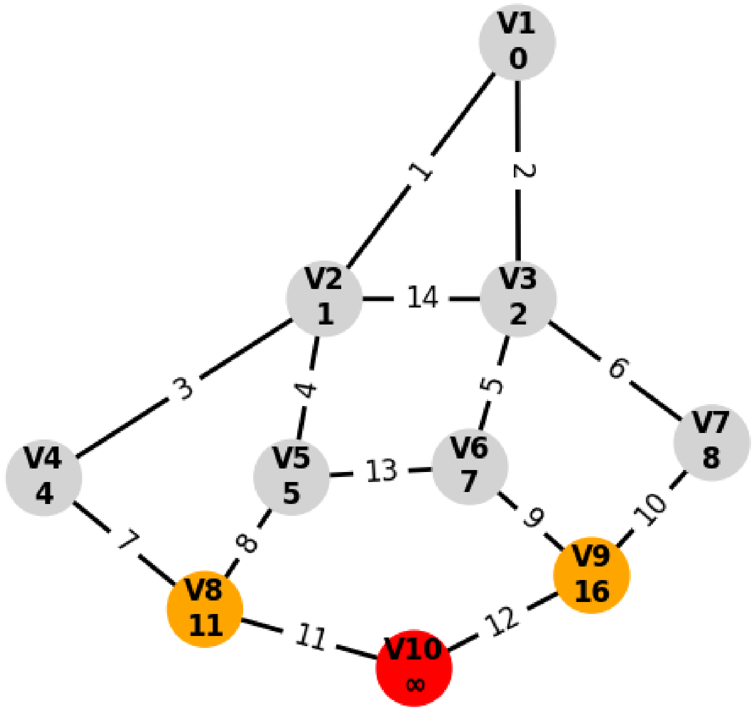
Queue: [(8, 'V7'), (11, 'V8'), (16, 'V9')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

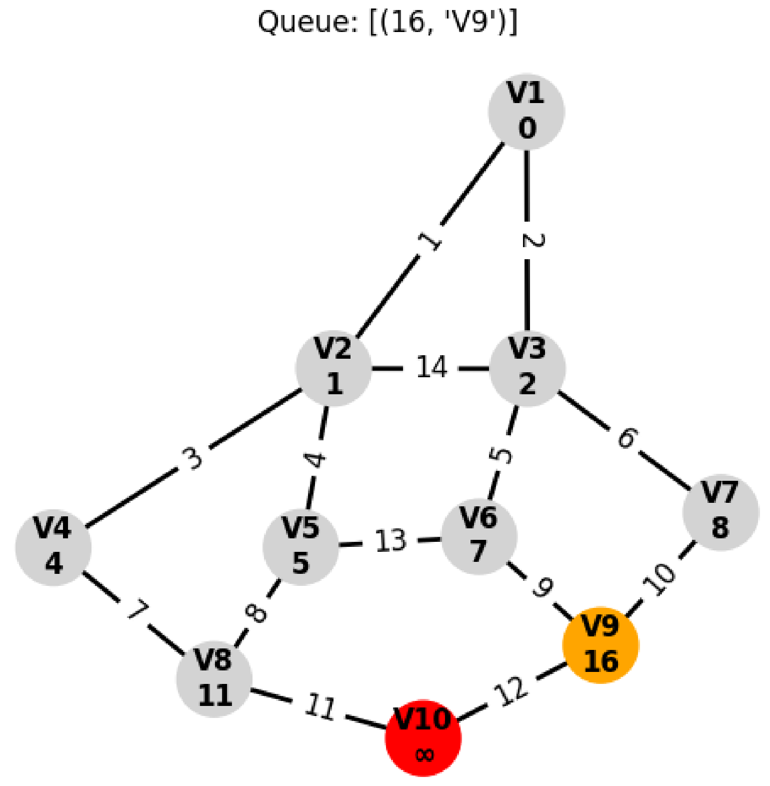
다익스트라 알고리즘 (Dijkstra Algorithm)

Queue: [(11, 'V8'), (16, 'V9')]



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

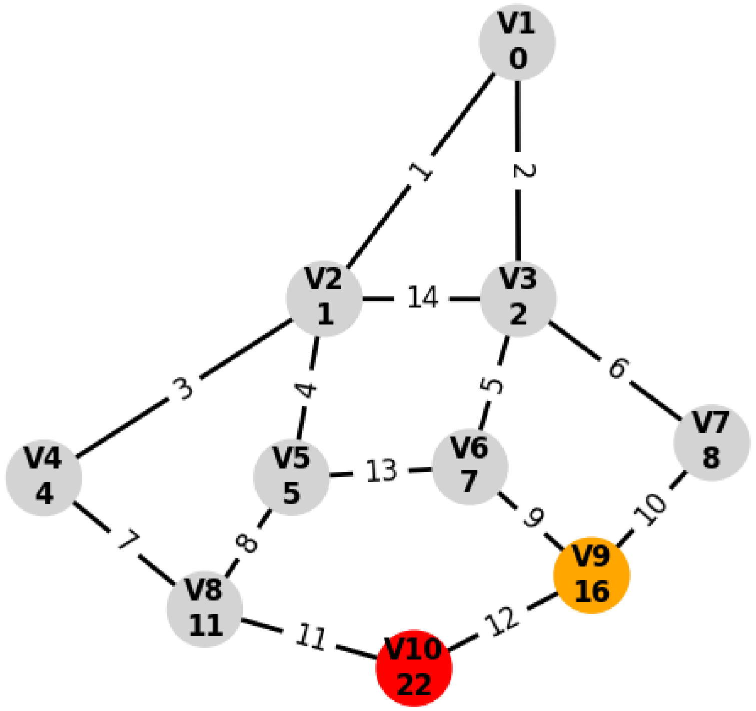
다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	∞	
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

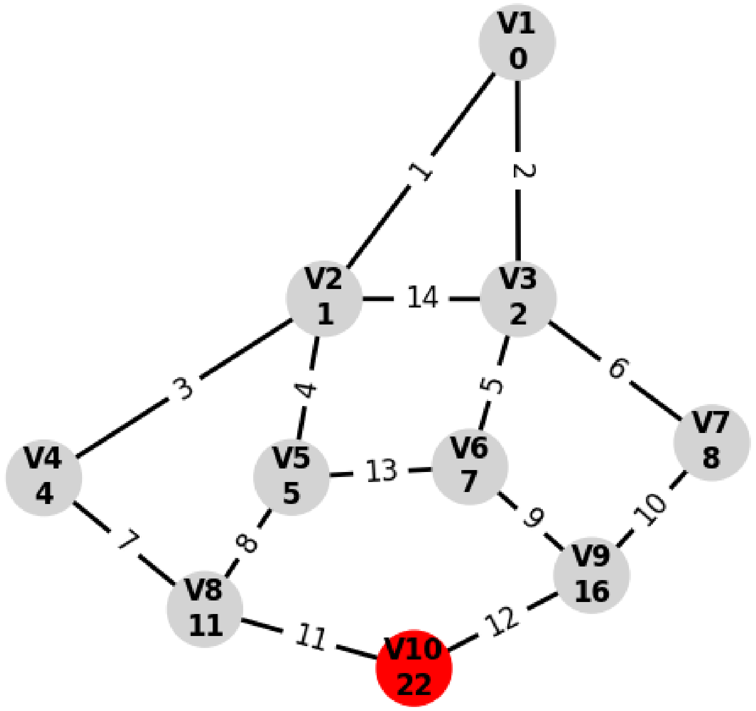
다익스트라 알고리즘 (Dijkstra Algorithm)

Queue: [(16, 'V9'), (22, 'V10')]



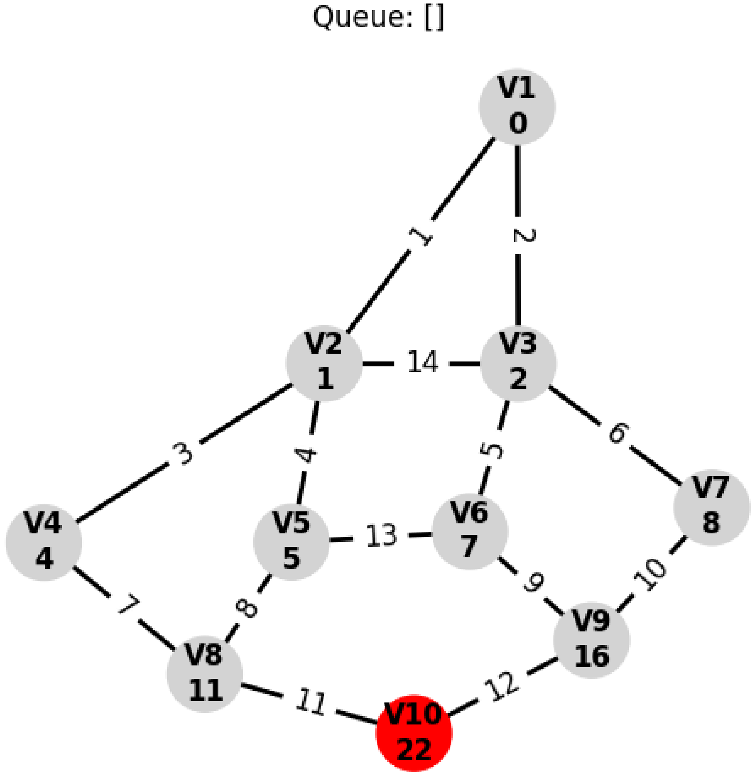
Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)



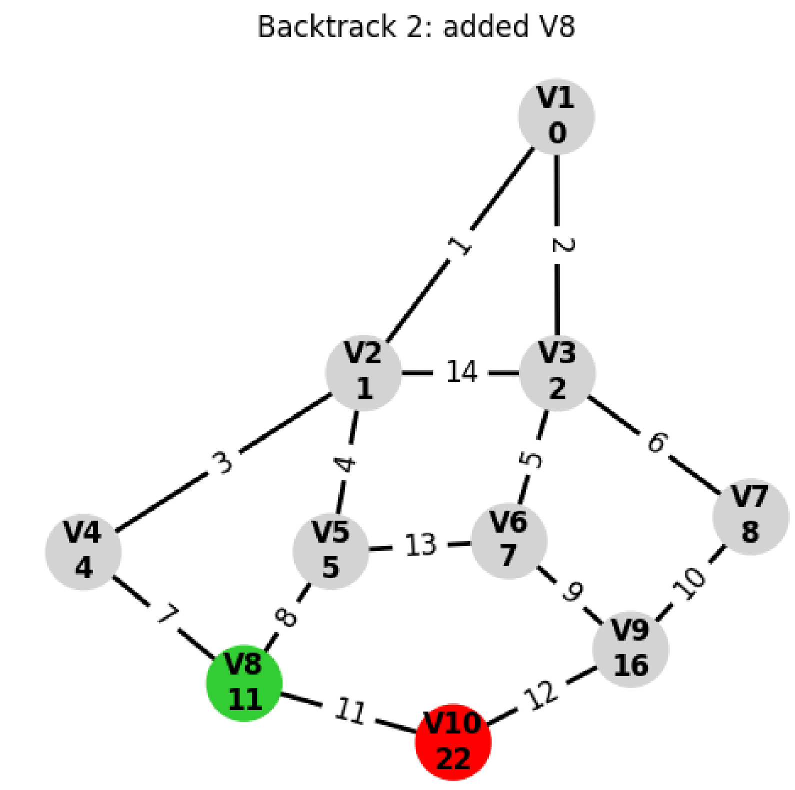
Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

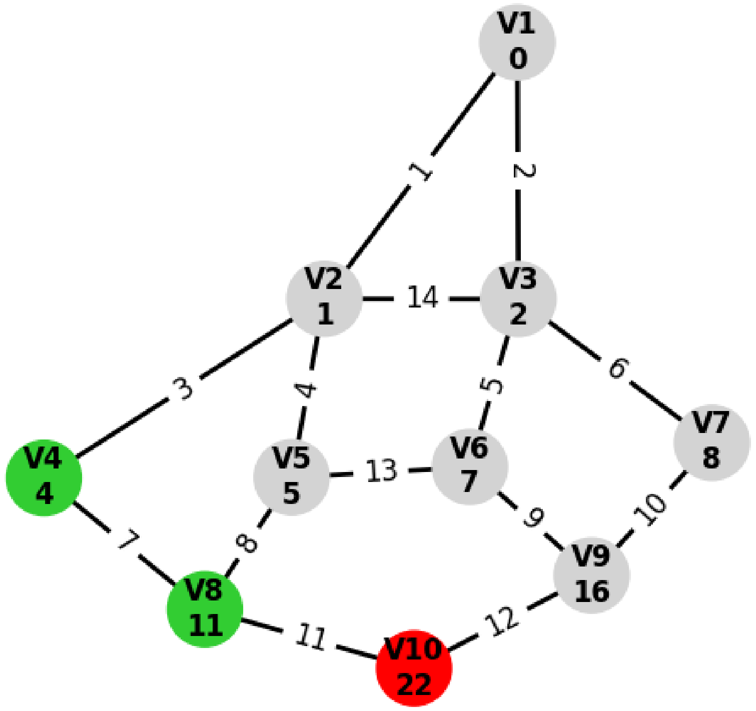
다익스트라 알고리즘 (Dijkstra Algorithm)



Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)

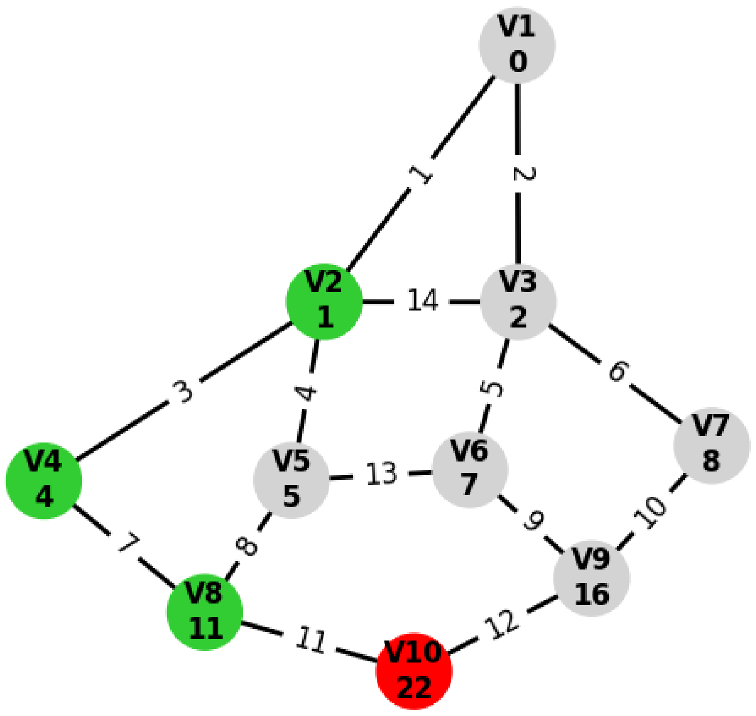
Backtrack 3: added V4



Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)

Backtrack 4: added V2



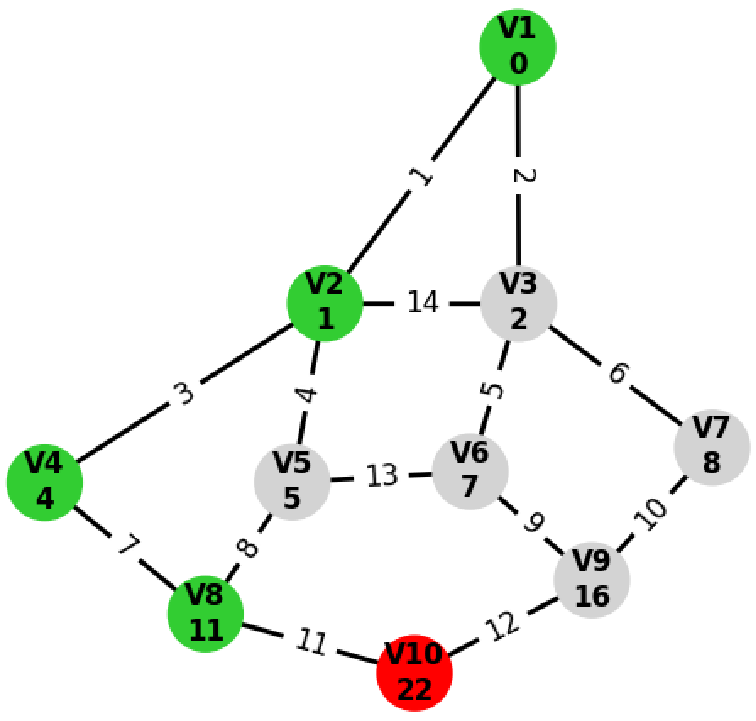
Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)

특징

- Priority Queue 사용
- 가중치가 양수인 경우 항상 최단경로를 찾을 수 있음
- 시간 복잡도: O((V+E) log V)
- 공간 복잡도: O(V+E)

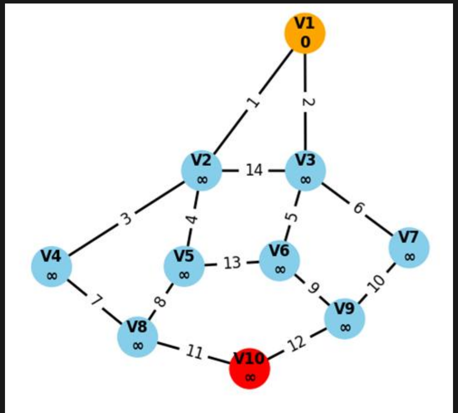
Backtrack 5: added V1



Node	Dist.	Pred.
V1	0	
V10	22	V8
V2	1	V1
V3	2	V1
V4	4	V2
V5	5	V2
V6	7	V3
V7	8	V3
V8	11	V4
V9	16	V6

다익스트라 알고리즘 (Dijkstra Algorithm)

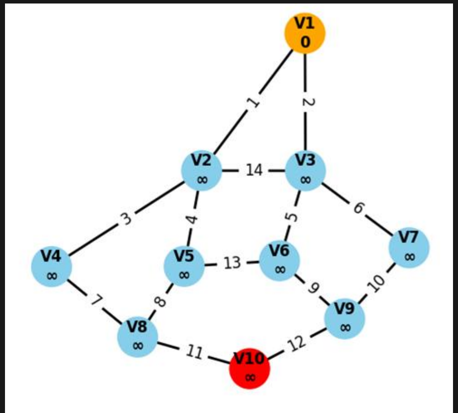
수도 코드 (pseudo code)




```python
G = nx.Graph()
nodes = [f"V{i}" for i in range(1, 11)]
G.add_nodes_from(nodes)
edges = [
    ("V1", "V2"), ("V1", "V3"), ("V2", "V4"), ("V2", "V5"),
    ("V3", "V6"), ("V3", "V7"), ("V4", "V8"), ("V5", "V8"),
    ("V6", "V9"), ("V7", "V9"), ("V8", "V10"), ("V9", "V10"),
    ("V5", "V6"), ("V2", "V3")
]
G.add_edges_from(edges)

# Assign default weight = 1 for each edge
for u, v in G.edges():
    G[u][v]['weight'] = 1

# Run Dijkstra
distances, predecessors = dijkstra(G, source="V1")
```

다익스트라 알고리즘 (Dijkstra Algorithm)

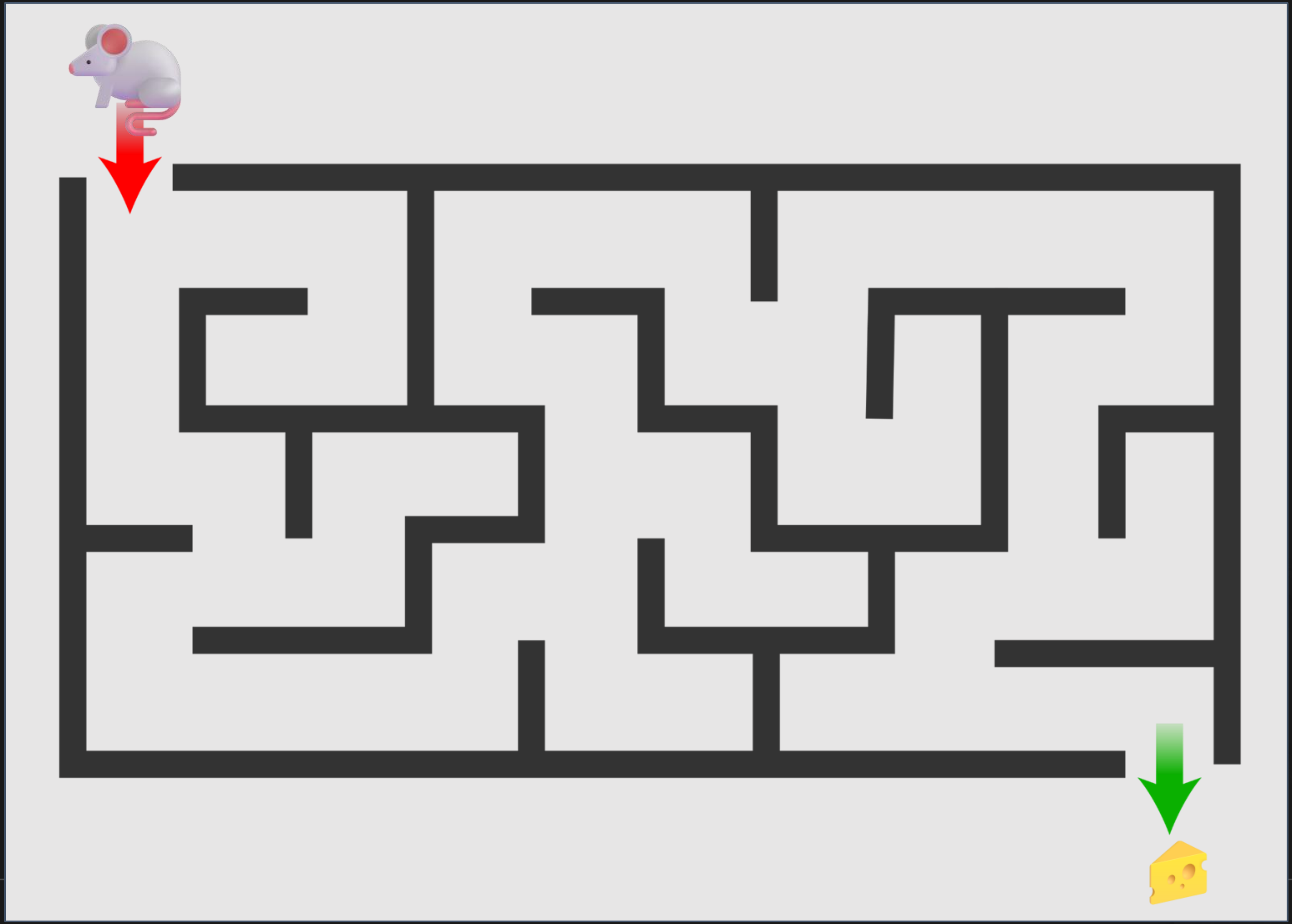
수도 코드 (pseudo code)




```
G = nx.Graph()
nodes = [f"V{i}" for i in range(1, 11)]
G.add_nodes_from(nodes)
edges = [
    ("V1", "V2"), ("V1", "V3"), ("V2", "V4"), ("V2", "V5"),
    ("V3", "V6"), ("V3", "V7"), ("V4", "V8"), ("V5", "V8"),
    ("V6", "V9"), ("V7", "V9"), ("V8", "V10"), ("V9", "V10"),
    ("V5", "V6"), ("V2", "V3")
]
G.add_edges_from(edges)

# Assign default weight = 1 for each edge
for u, v in G.edges():
    G[u][v]['weight'] = 1

# Run Dijkstra
distances, predecessors = dijkstra(G, source="V1")
```

```
import heapq

def dijkstra(graph, source):

    dist = {node: float('inf') for node in graph}
    prev = {node: None for node in graph}
    dist[source] = 0

    pq = [(0, source)]

    while pq:
        current_dist, u = heapq.heappop(pq)

        if current_dist > dist[u]:
            continue

        for v, weight in graph[u].items():
            new_dist = current_dist + weight
            if new_dist < dist[v]:
                dist[v] = new_dist
                prev[v] = u
                heapq.heappush(pq, (new_dist, v))

    return dist, prev
```

A* 알고리즘



A* 알고리즘

휴리스틱 (Heuristic)

- 목표 지점까지의 거리를 근사
- Admissible: 실제 최단 경로 비용보다 과대평가(overestimate) 하지 않아야 함.

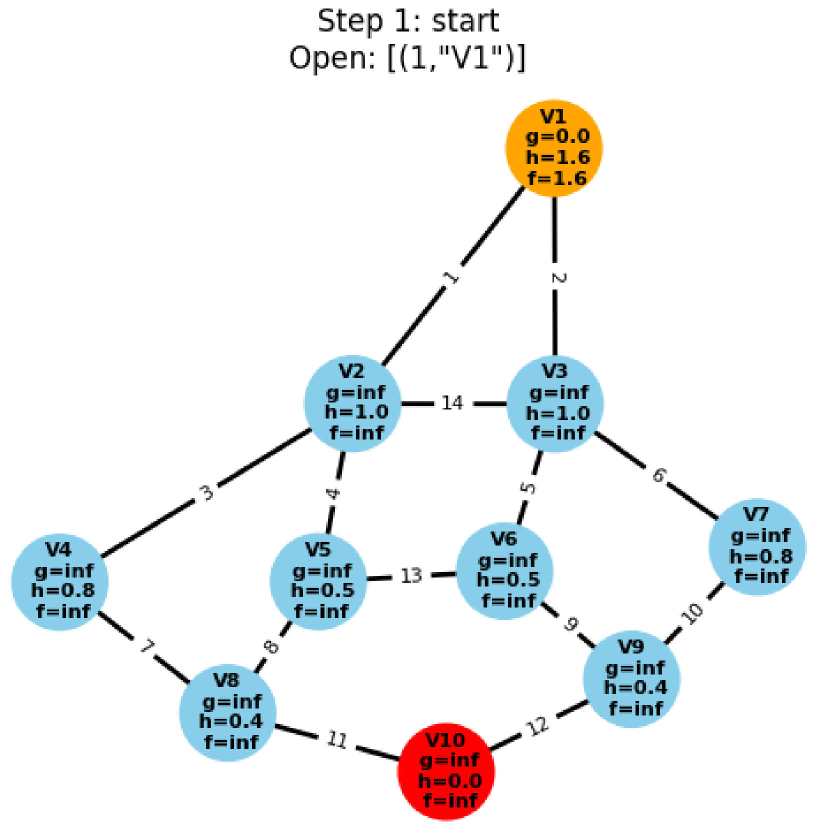
$$h(n) \leq \text{실제 최단 거리}(n \to \text{goal})$$

- Consistency: 휴리스틱에 의해 선택된 노드는 실제로 목표 지점에 가까워지고 있어야 함.

$$h(n) \leq w(n, m) + h(m)$$

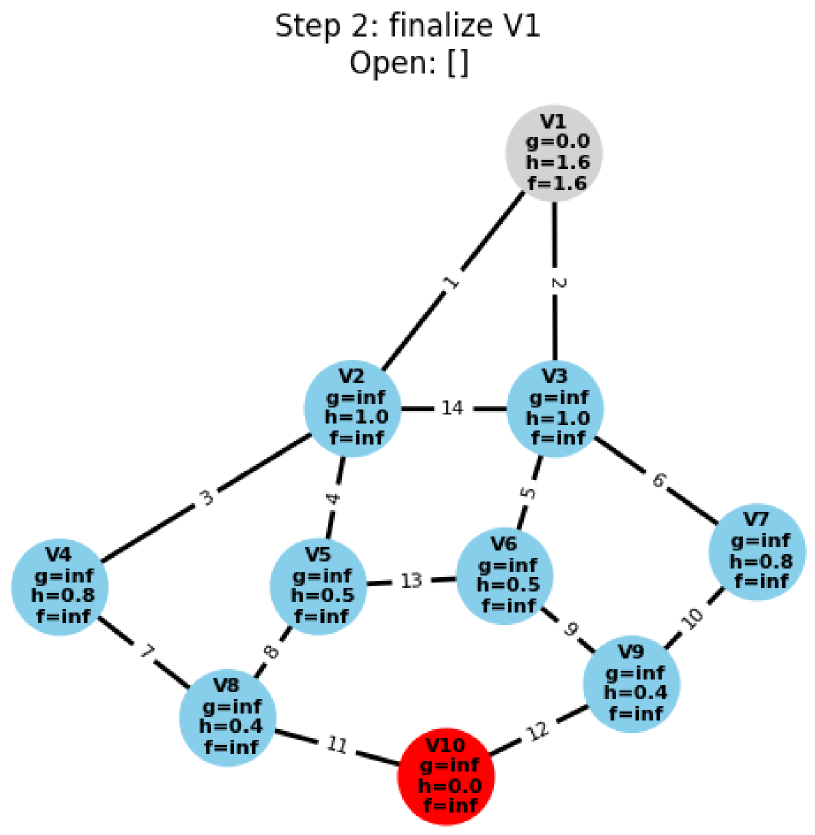
- 예시: Manhattan Distance

A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	∞	1.0	∞	
V3	∞	1.0	∞	
V4	∞	0.8	∞	
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

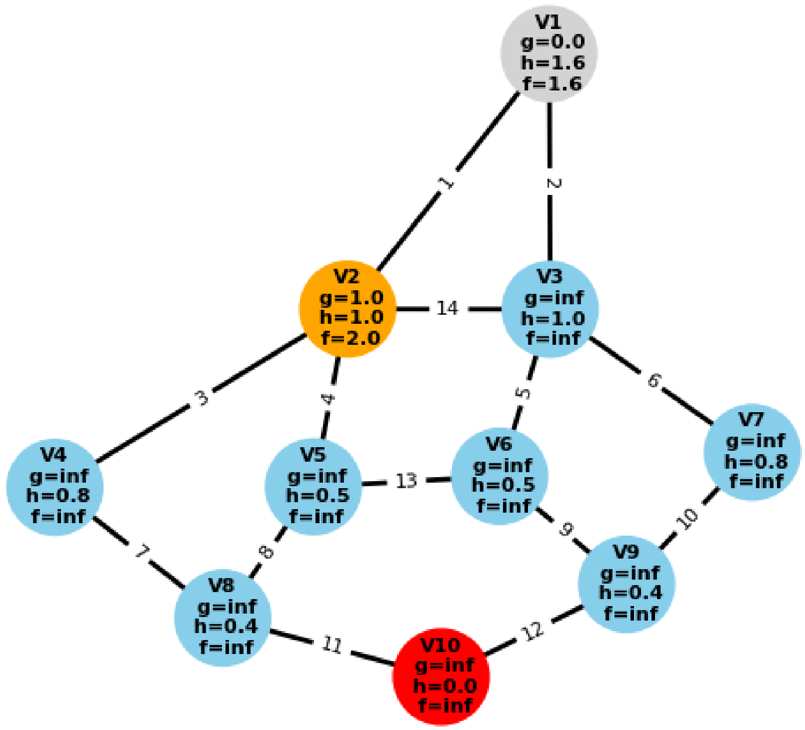
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	∞	1.0	∞	
V3	∞	1.0	∞	
V4	∞	0.8	∞	
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

A* 알고리즘

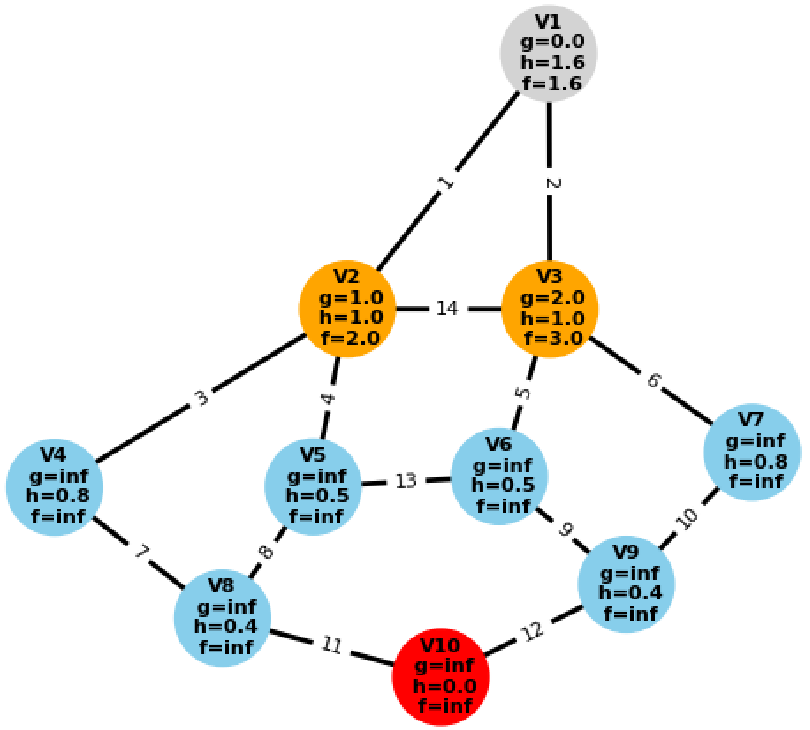
Step 3: update V2 via V1
Open: [(1,"V2")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	∞	1.0	∞	
V4	∞	0.8	∞	
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

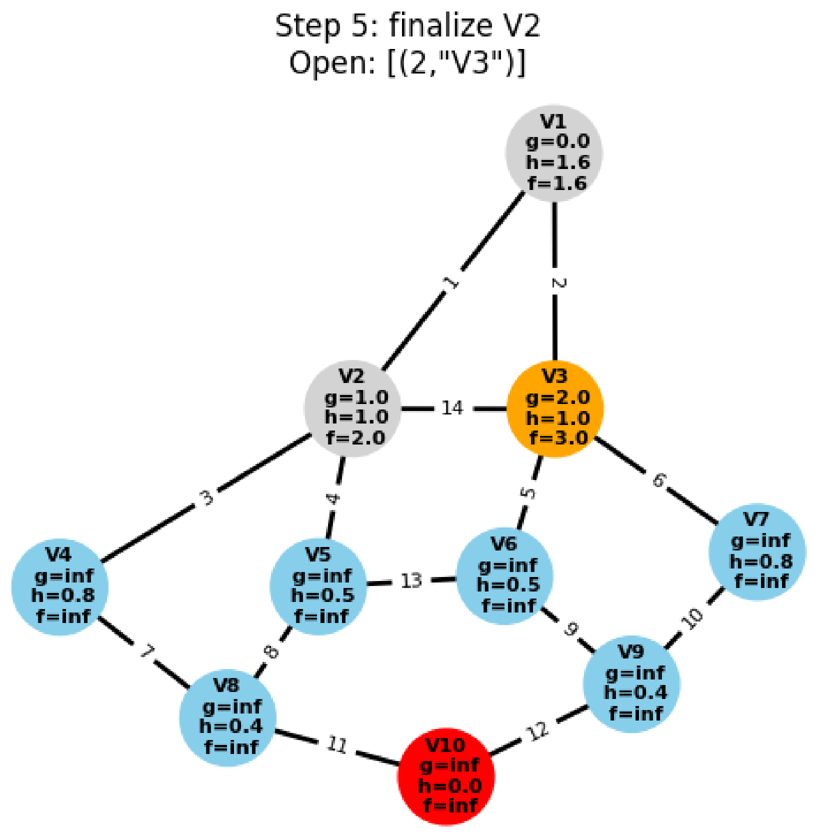
A* 알고리즘

Step 4: update V3 via V1
Open: [(1,"V2"), (2,"V3")]



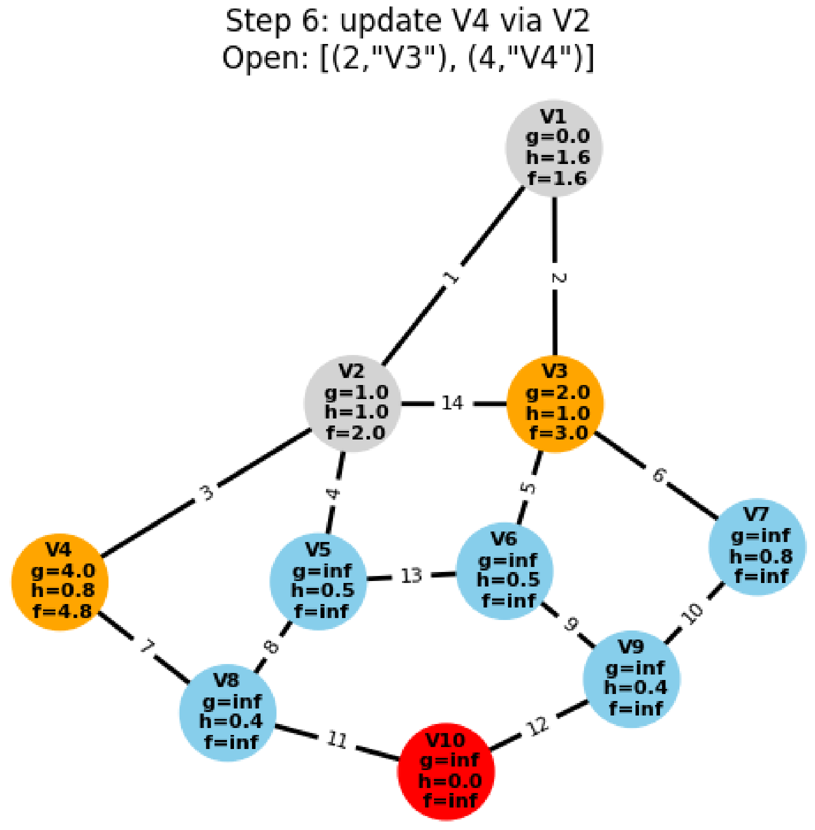
Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	∞	0.8	∞	
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	∞	0.8	∞	
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

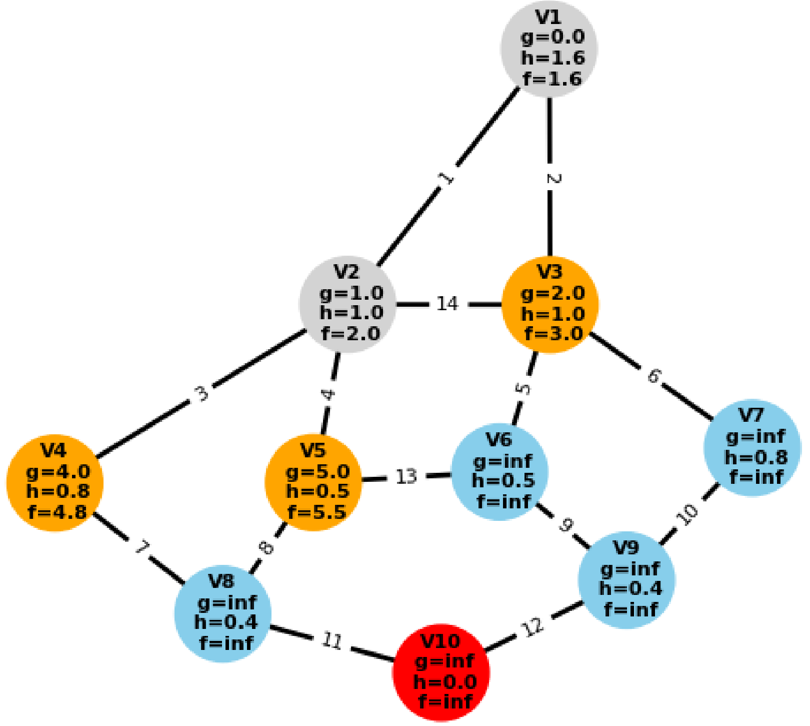
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	∞	0.5	∞	
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

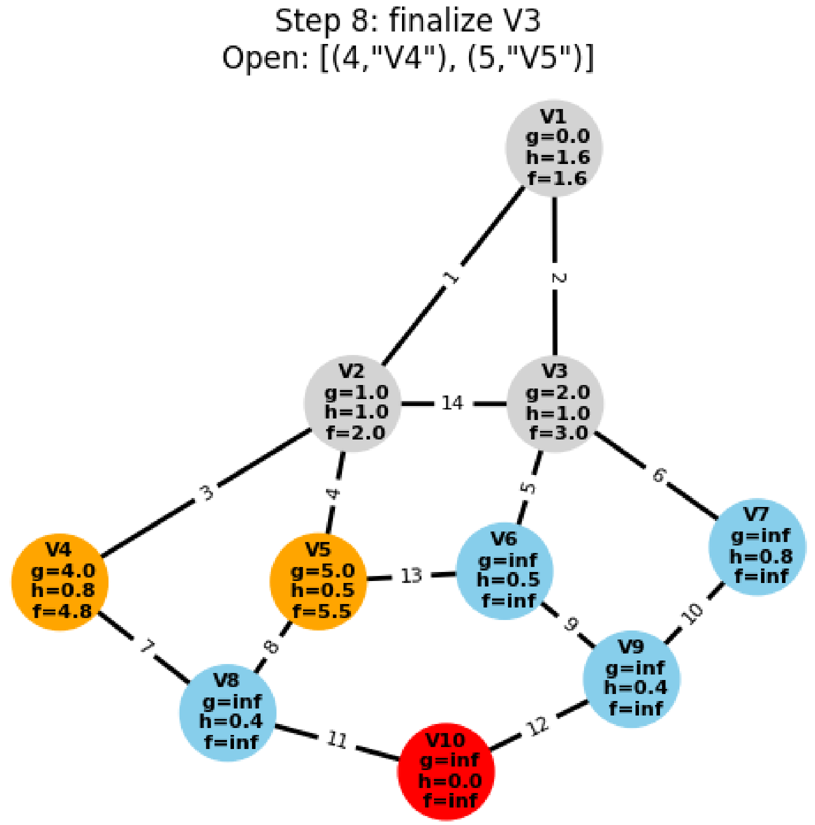
A* 알고리즘

Step 7: update V5 via V2
Open: [(2,"V3"), (4,"V4"), (5,"V5")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

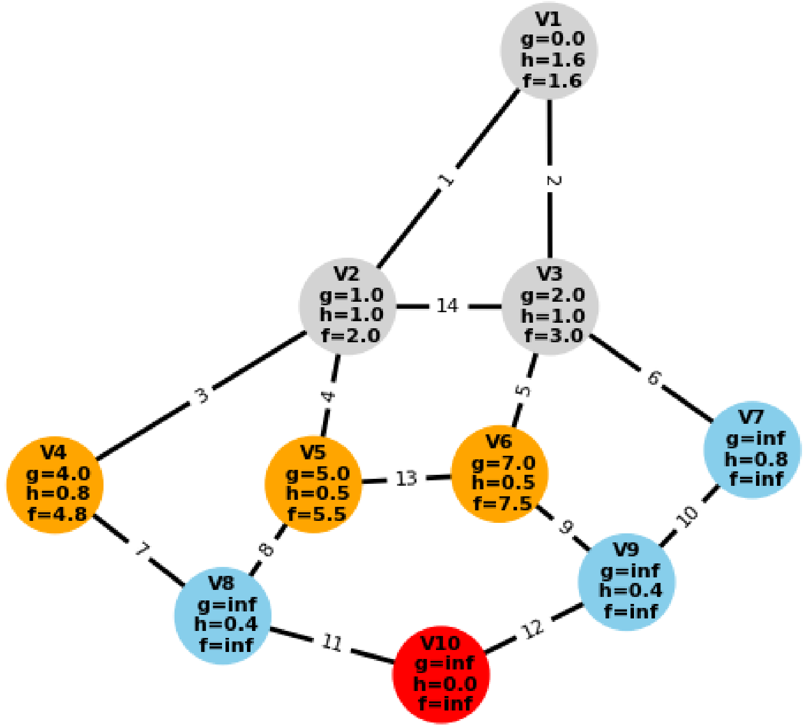
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	∞	0.5	∞	
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

A* 알고리즘

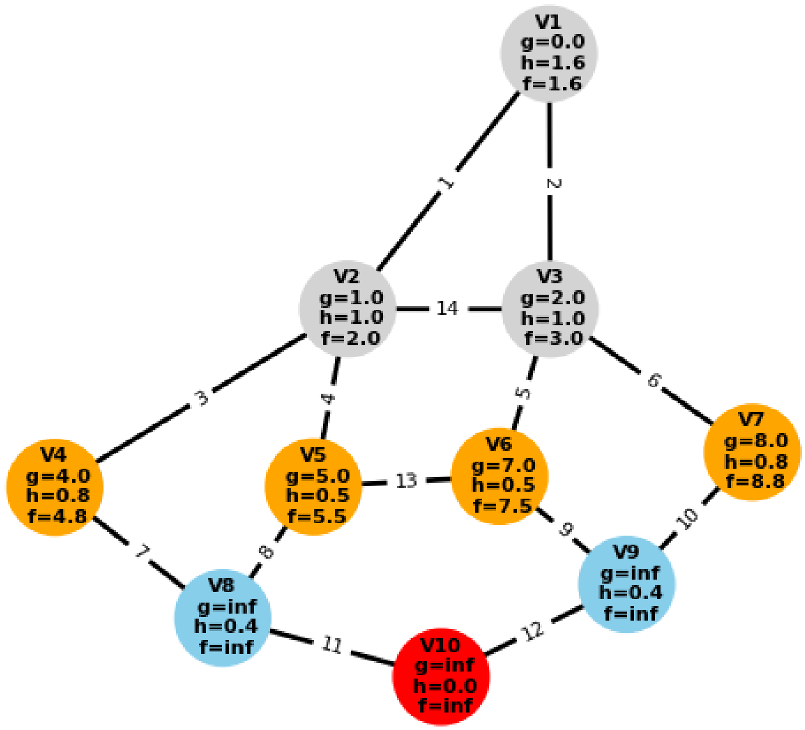
Step 9: update V6 via V3
Open: [(4,"V4"), (5,"V5"), (7,"V6")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	∞	0.8	∞	
V8	∞	0.4	∞	
V9	∞	0.4	∞	

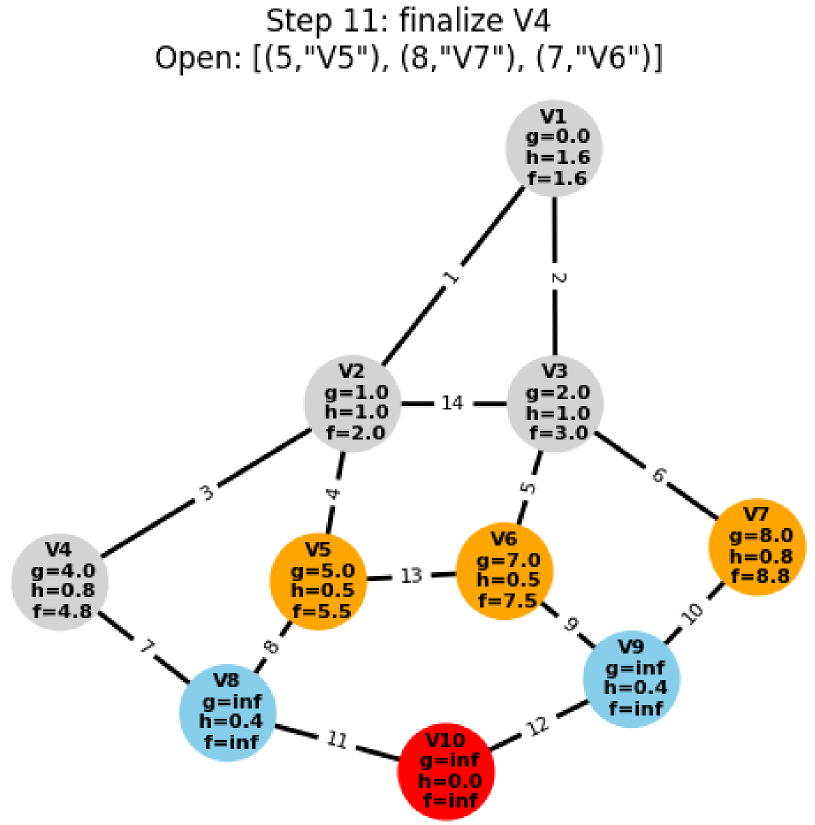
A* 알고리즘

Step 10: update V7 via V3
Open: [(4,"V4"), (5,"V5"), (7,"V6"), (8,"V7")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	∞	0.4	∞	
V9	∞	0.4	∞	

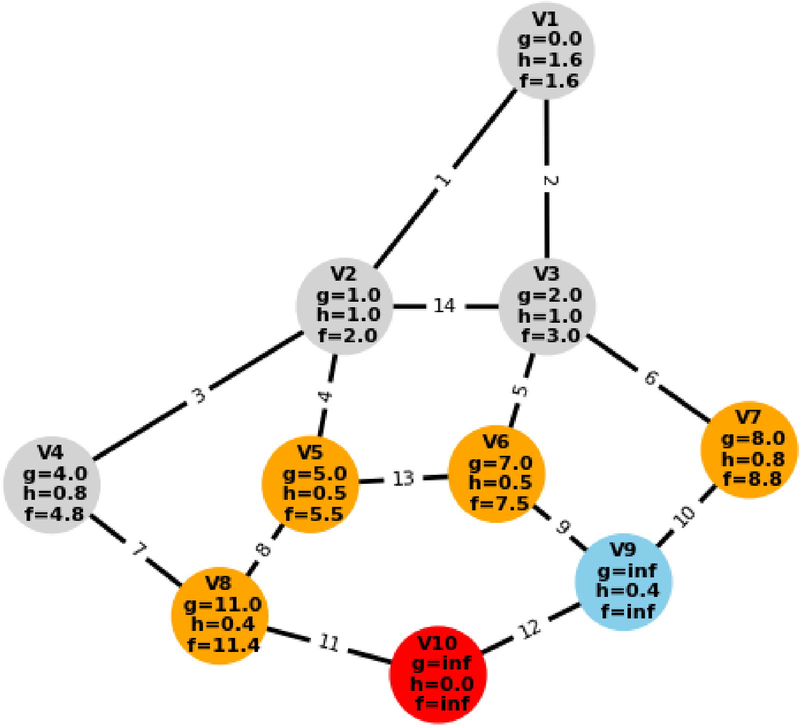
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	∞	0.4	∞	
V9	∞	0.4	∞	

A* 알고리즘

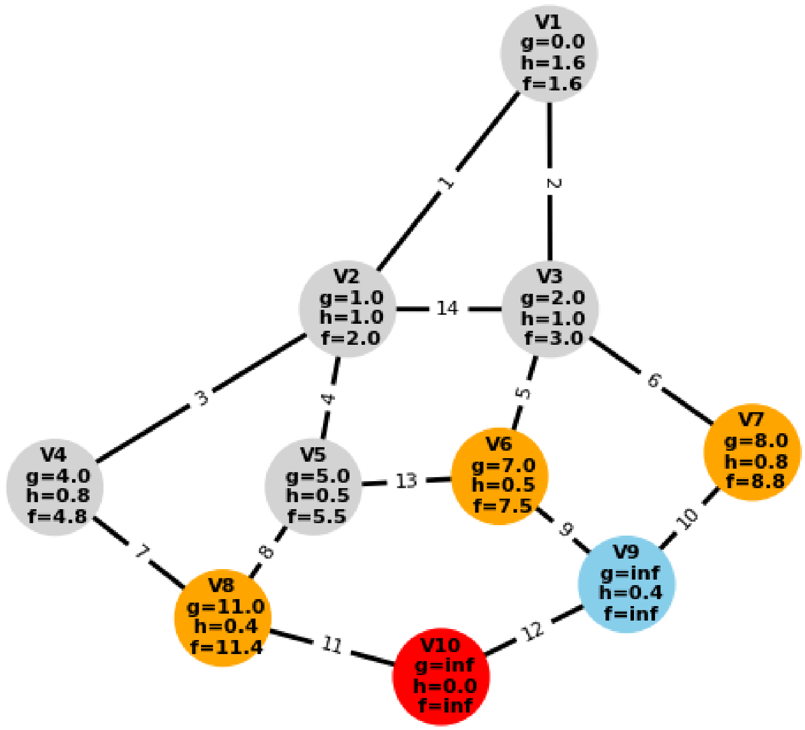
Step 12: update V8 via V4
Open: [(5,"V5"), (8,"V7"), (7,"V6"), (11,"V8")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	∞	0.4	∞	

A* 알고리즘

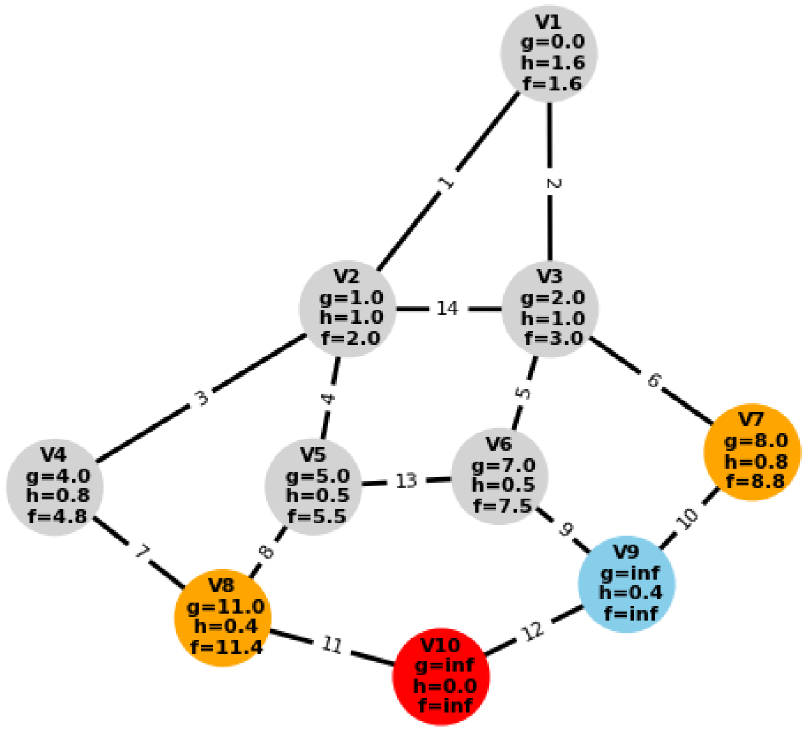
Step 13: finalize V5
Open: [(7,"V6"), (8,"V7"), (11,"V8")]



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	∞	0.4	∞	

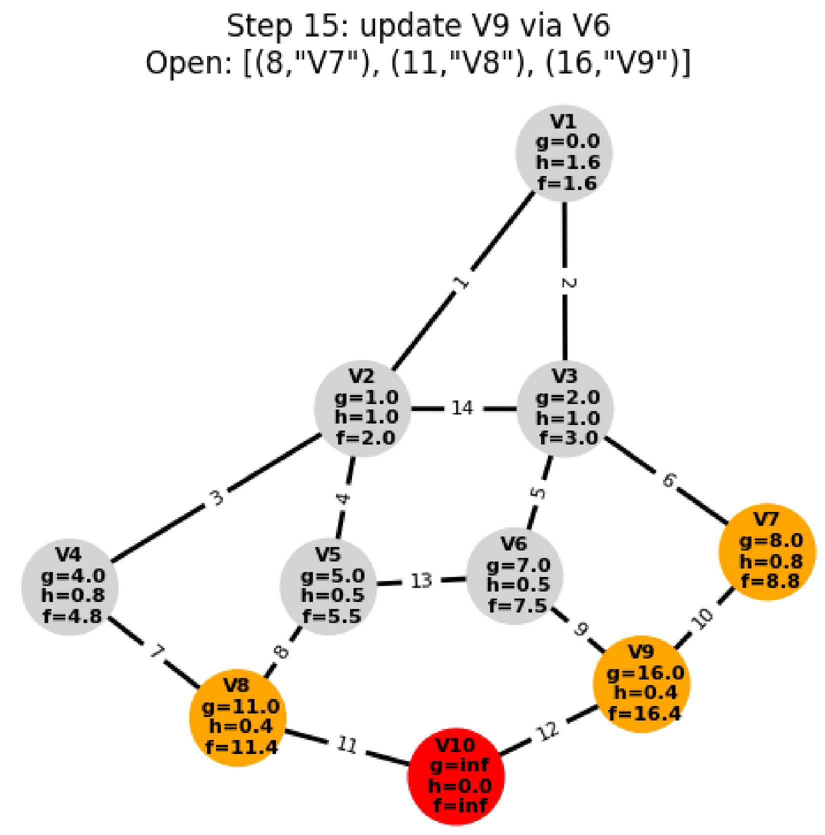
A* 알고리즘

Step 14: finalize V6
Open: [(8,"V7"), (11,"V8")]



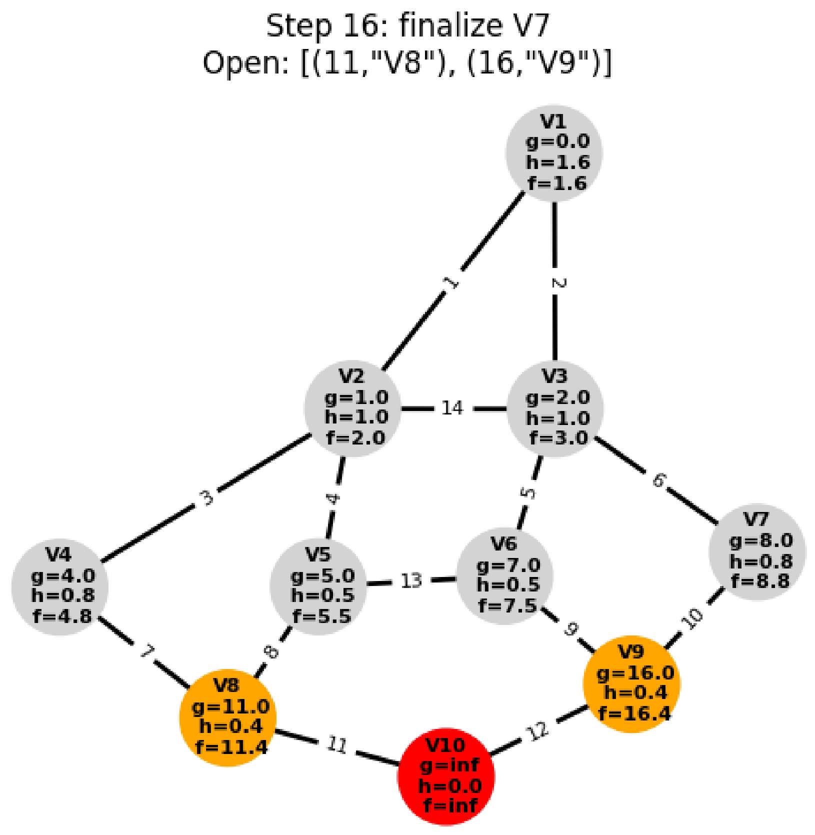
Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	∞	0.4	∞	

A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

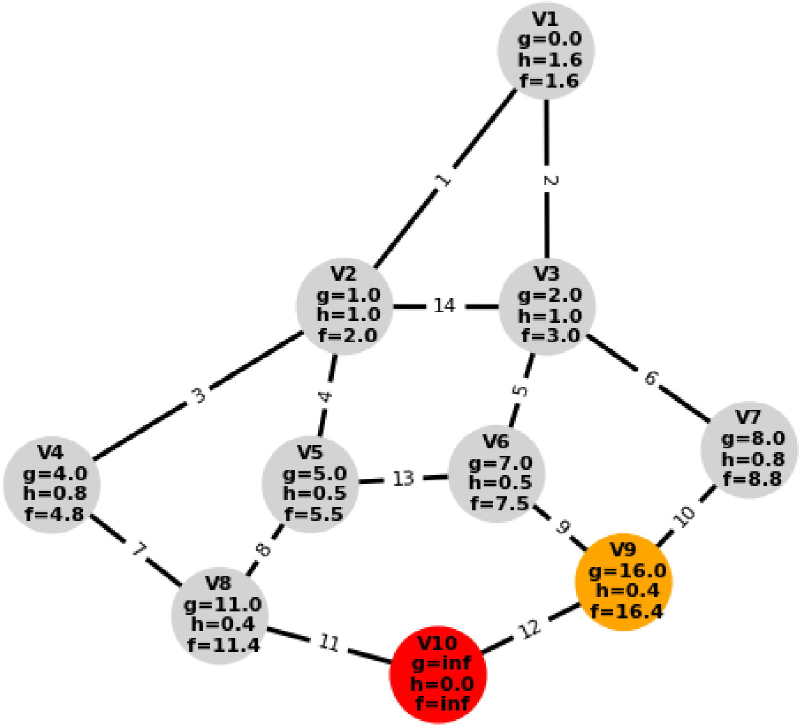
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

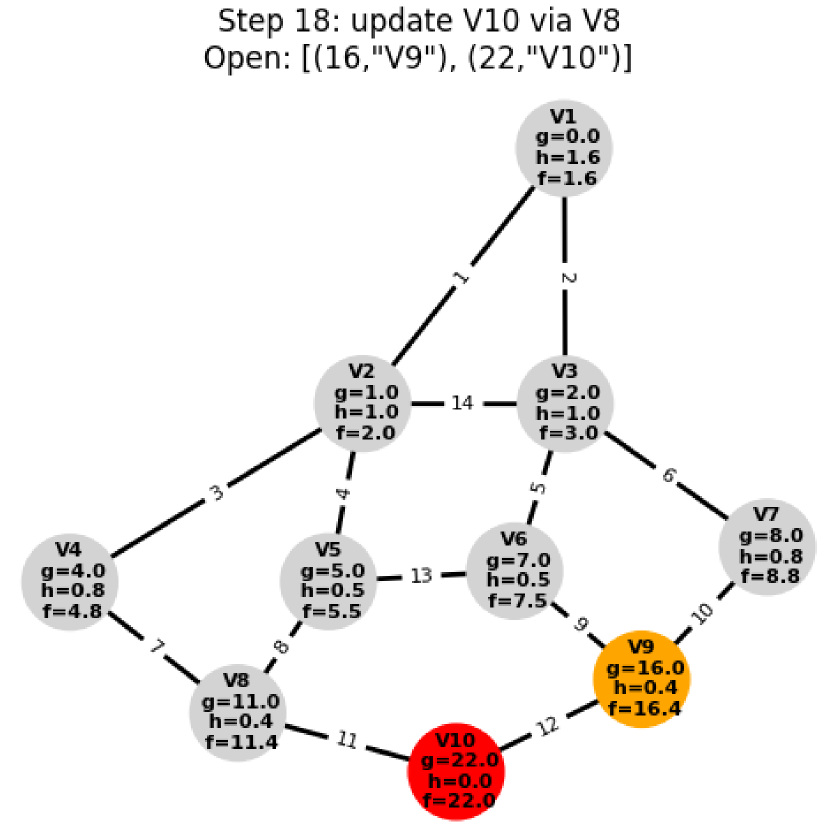
A* 알고리즘

Step 17: finalize V8
Open: [(16,"V9")]



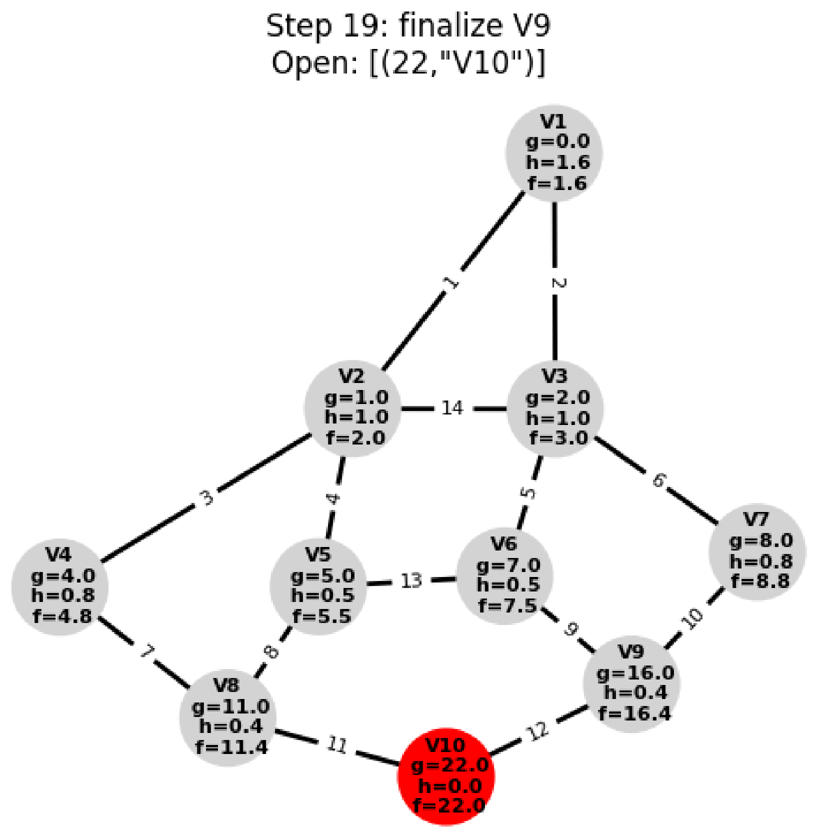
Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	∞	0.0	∞	
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

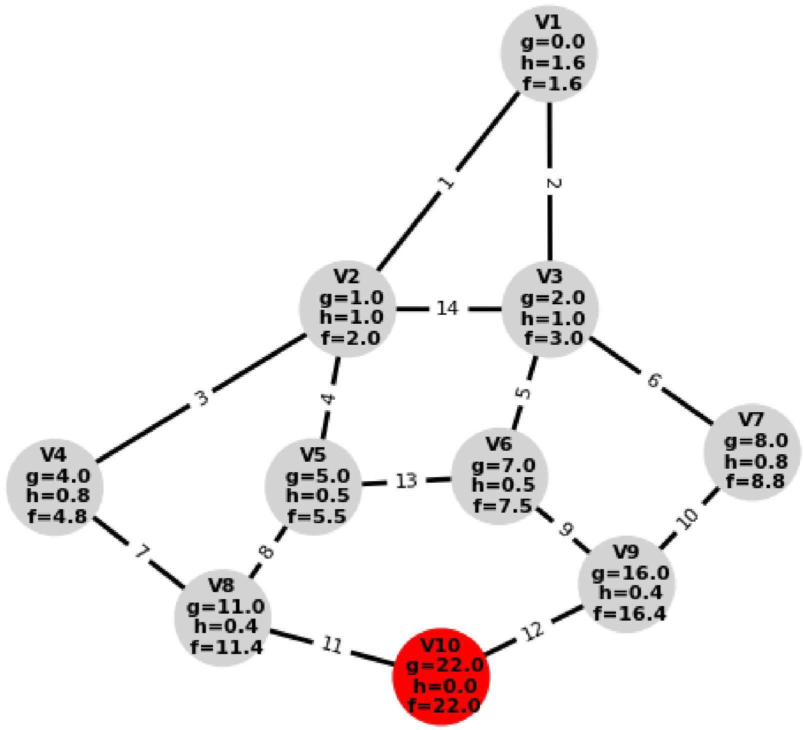
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

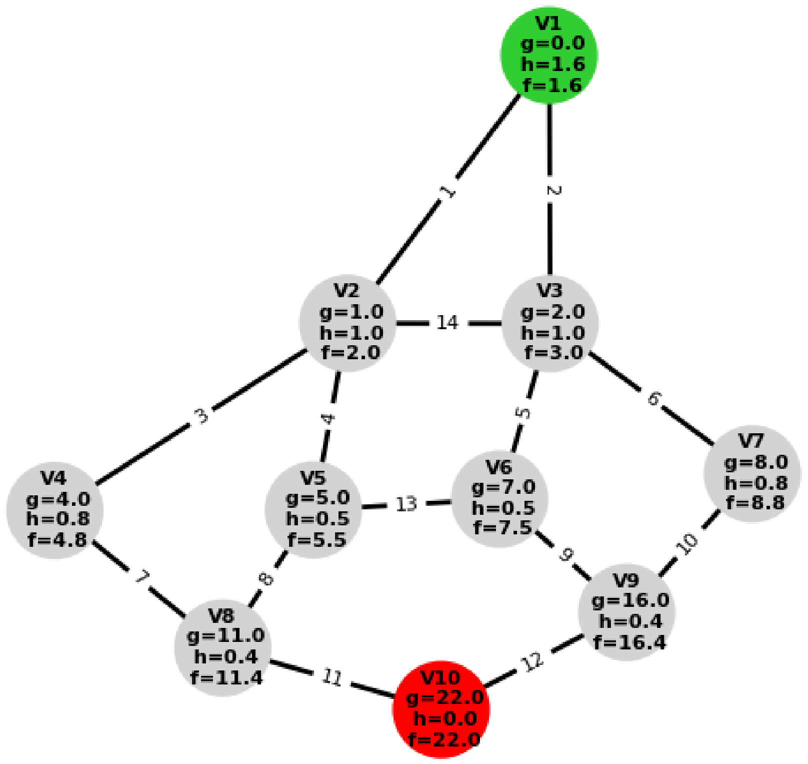
Step 20: finalize V10
Open: []



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

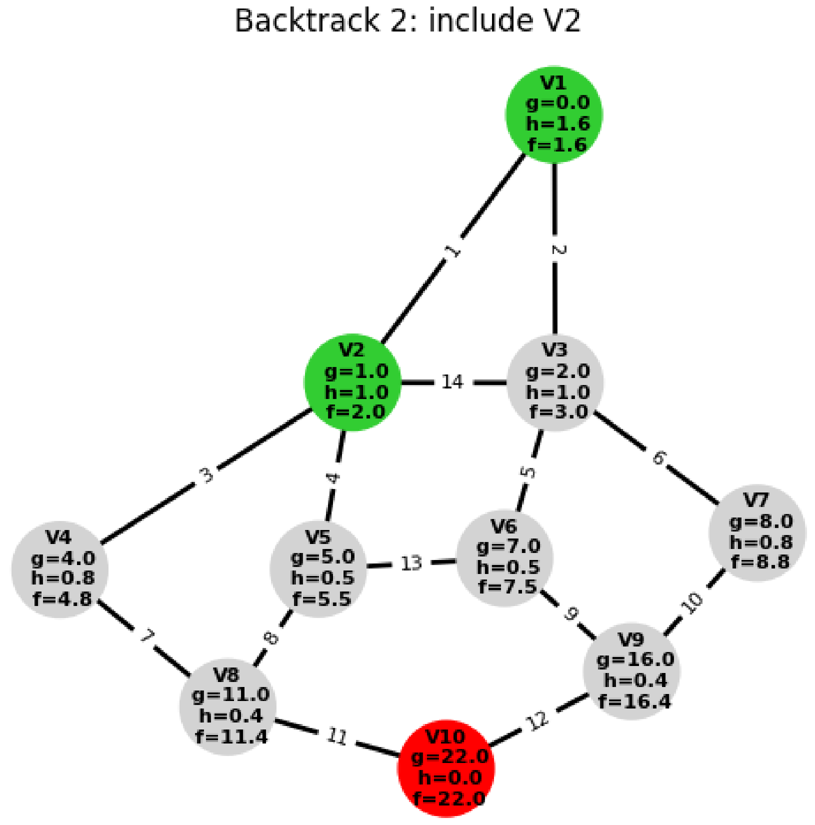
A* 알고리즘

Backtrack 1: include V1



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

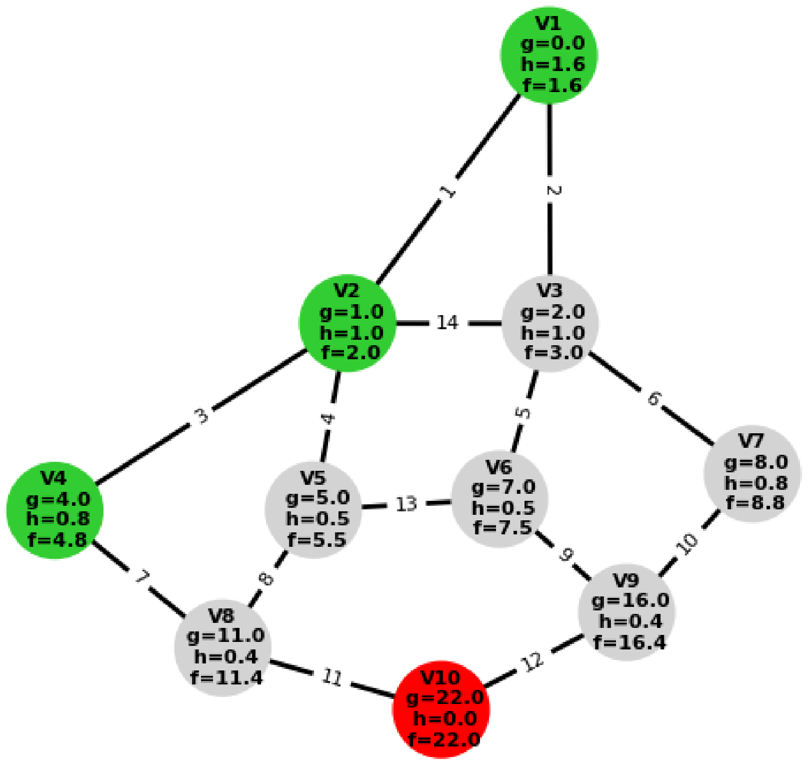
A* 알고리즘



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

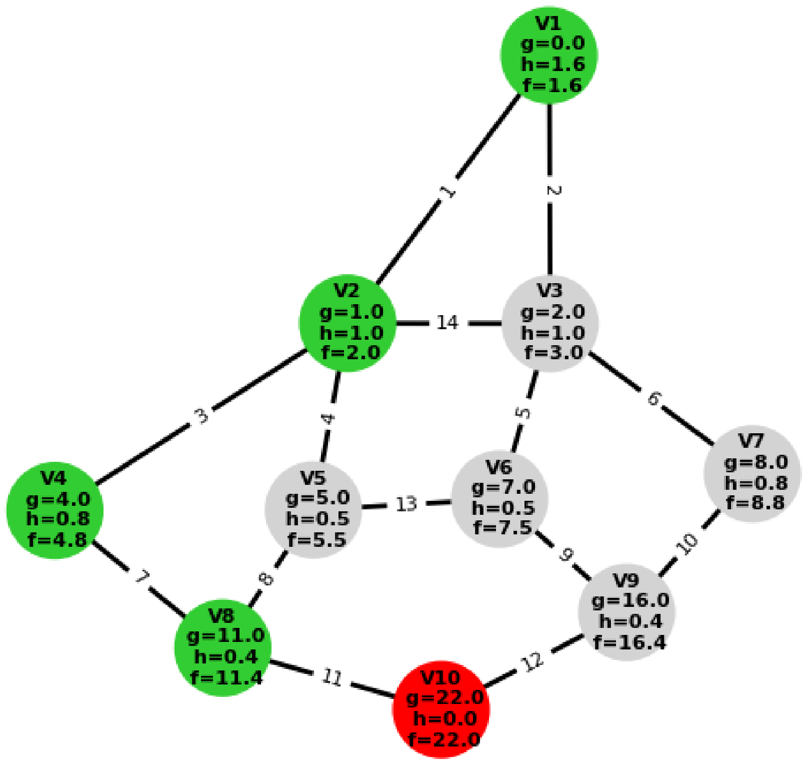
Backtrack 3: include V4



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

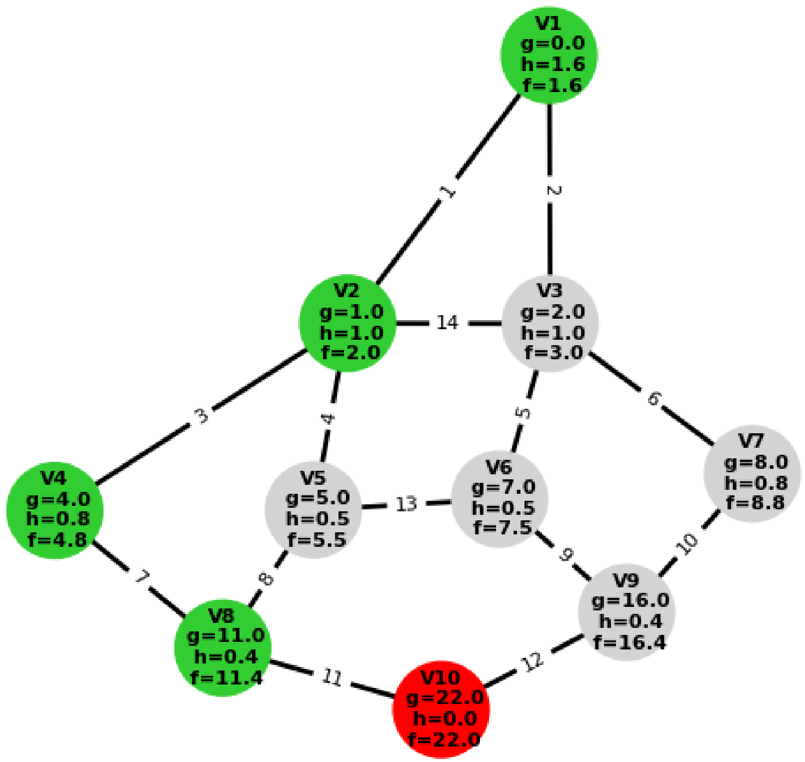
Backtrack 4: include V8



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

Backtrack 5: include V10



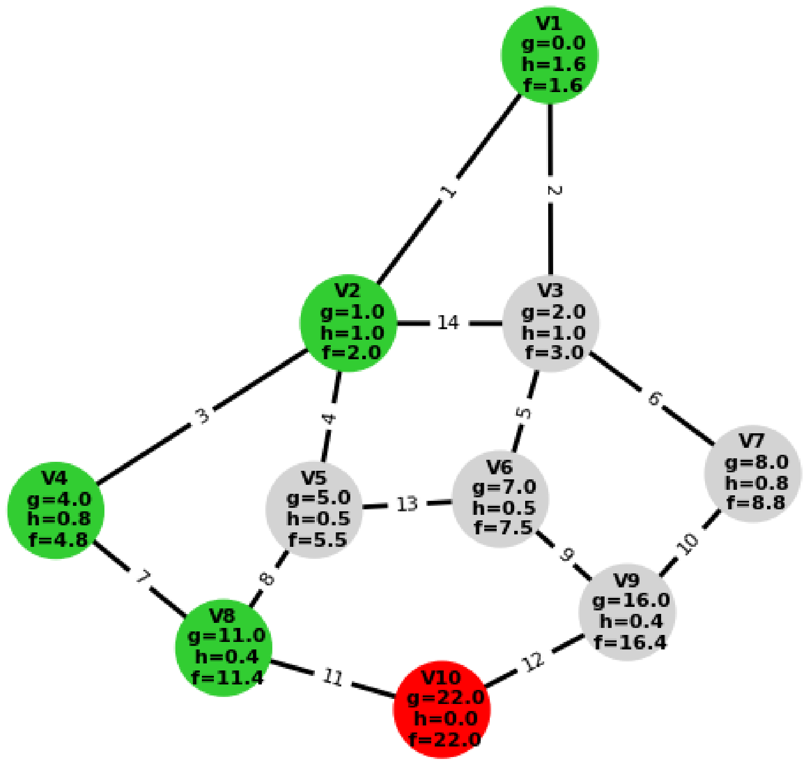
Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

특징

- Priority Queue 사용
- 가중치가 양수인 경우 항상 최단경로를 찾을 수 있음
- 시간 복잡도와 공간 복잡도가 휴리스틱의 성능에 따라서 크게 달라짐

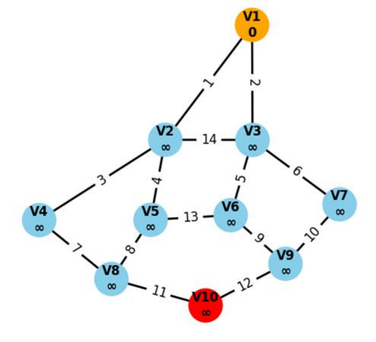
Backtrack 5: include V10



Node	g-score	h-score	f-score	Parent
V1	0.0	1.6	1.6	
V10	22.0	0.0	22.0	V8
V2	1.0	1.0	2.0	V1
V3	2.0	1.0	3.0	V1
V4	4.0	0.8	4.8	V2
V5	5.0	0.5	5.5	V2
V6	7.0	0.5	7.5	V3
V7	8.0	0.8	8.8	V3
V8	11.0	0.4	11.4	V4
V9	16.0	0.4	16.4	V6

A* 알고리즘

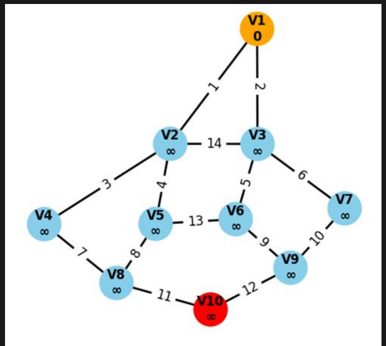
수도 코드 (pseudo code)




```
G = nx.Graph()
nodes = [f"V{i}" for i in range(1, 11)]
G.add_nodes_from(nodes)
edges = [
    ("V1", "V2"), ("V1", "V3"), ("V2", "V4"), ("V2", "V5"),
    ("V3", "V6"), ("V3", "V7"), ("V4", "V8"), ("V5", "V8"),
    ("V6", "V9"), ("V7", "V9"), ("V8", "V10"), ("V9", "V10"),
    ("V5", "V6"), ("V2", "V3")
]
G.add_edges_from(edges)

for i, (u, v) in enumerate(edges, start=1):
    G[u][v]['weight'] = i

pos = nx.spring_layout(G, seed=42)

def heuristic(n):
    return math.dist(pos[n], pos["V10"])

path = astar(G, start="V1", goal="V10", h=heuristic)
print("Found path:", path)
```

A* 알고리즘

수도 코드 (pseudo code)




```python
G = nx.Graph()
nodes = [f"V{i}" for i in range(1, 11)]
G.add_nodes_from(nodes)
edges = [
    ("V1", "V2"), ("V1", "V3"), ("V2", "V4"), ("V2", "V5"),
    ("V3", "V6"), ("V3", "V7"), ("V4", "V8"), ("V5", "V8"),
    ("V6", "V9"), ("V7", "V9"), ("V8", "V10"), ("V9", "V10"),
    ("V5", "V6"), ("V2", "V3")
]
G.add_edges_from(edges)

for i, (u, v) in enumerate(edges, start=1):
    G[u][v]['weight'] = i

pos = nx.spring_layout(G, seed=42)

def heuristic(n):
    return math.dist(pos[n], pos["V10"])

path = astar(G, start="V1", goal="V10", h=heuristic)
print("Found path:", path)
```

```python
def astar(G, start, goal, h):
    open_set   = {start}
    closed_set = set()
    g_score    = {v: float('inf') for v in G.nodes()}
    f_score    = {v: float('inf') for v in G.nodes()}
    parent     = {}

    g_score[start] = 0
    f_score[start] = h(start)

    while open_set:
        current = min(open_set, key=lambda v: f_score[v])

        if current == goal:
            return reconstruct_path(parent, current)

        open_set.remove(current)
        closed_set.add(current)

        for neighbor in G.neighbors(current):
            if neighbor in closed_set:
                continue

            cost = G[current][neighbor].get('weight', 1)
            tentative_g = g_score[current] + cost

            if tentative_g < g_score[neighbor]:
                parent[neighbor]    = current
                g_score[neighbor]  = tentative_g
                f_score[neighbor]  = tentative_g + h(neighbor)
                if neighbor not in open_set:
                    open_set.add(neighbor)

    return None
```

```python
def reconstruct_path(parent, node):
    path = [node]
    while node in parent:
        node = parent[node]
        path.append(node)
    return path[::-1]
```

Dijkstra vs. A*

Dijkstra Algorithm

- Priority Queue (FIFO) 사용
- 시간 복잡도: O((V+E) log V)
- 공간 복잡도: O(V+E)
- 가중치가 양수인 경우 항상 최단경로를 찾을 수 있음

최단 경로 알고리즘

A* Algorithm

- Priority Queue (FIFO) 사용
- 가중치가 양수인 경우 항상 최단경로를 찾을 수 있음
- 시간 복잡도와 공간 복잡도가 휴리스틱의 성능에 따라서 크게 달라짐

강의 요약

01

Priority Queue

02

Dijkstra Algorithm

03

휴리스틱 (Heuristic)

Admissible

Consistency

04

A* Algorithm