201201871 서현택

1.목표

최단경로 알고리즘과 최소비용 신장트리 알고리즘을 통한 문제 해결

2.과제를 해결하는 방법

1) 다익스트라 알고리즘에 대한 이해
2) 프림 알고리즘에 대한 이해
3) 그래프에 대한 이해

3.과제를 해결한 방법

1) Dijkstra

반복을 통해 dist, prev, qnode를 초기화한다. 그리고 시작 노드의 dist 값을 0으로 설정한다. 모든 노드를 방문할 때까지 반복하며 최소 거리값과 그 값을 가지는 노드를 udist와 unode에 저장한다. 그 후 qnode에서 unode를 제거하고 wdgraph[unode]를 반복하면서, dist[unode]와 wdgraph[unode][nbr] 값이 저장된 alt와 dist를 비교하여 alt가 작다면 dist와 prev에 alt와 unode를 저장한다.
그 후, dist와 prev를 반환한다.

2) Prim

wugraph의 키 값을 반복하며 mst를 초기화하고, mst를 반복하며 remain_node에 node를 저장한다. start_node 값에 랜덤한 값을 저장하고, added_node에 start_node를 추가하고, remain_node에서 start_node를 삭제한다. 그 후 remain_node가 없어질 때까지, 반복을 하며 mst의 값을 설정한다. 마지막으로 mst를 반환한다.

3) Search_Min

src_node,와 dst_node을 None, dist를 float('inf')으로 초기화하고, added_node와 remained_node를 반복하면서 wugraph에서 해당 weight 값과 dist를 비교하여 dist가 더 크다면 dist를 wugraph의 weight 값으로 바꾸고, src_node와 dst_node를 src와 dst로 바꿔서 최소값을 저장한다. 마지막으로 src_node와 dst_node, dist를 반환한다.

4) Dijkstra Main

key1과 key2를 통해 WDGRAPH와 WDGRAPH[key1]를 반복하면서 WDGRAPH의 값을 graph의 엣지 값으로 추가하고, WDGRAPH에 저장된 weight 값을 graph와, data에 저장한다.
list형 변수 path를 만들고, prev에 dijkstra의 두 번째 열값을 저장하여 이전 엣지 값을 저장한다. 그 후, prev를 반복하며 각 엣지의 prev값과 현재 엣지의 값을 path의 저장하여 경로를 나타낸다.

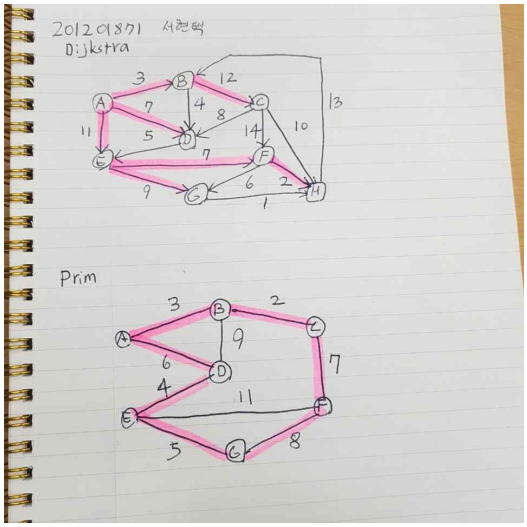
5) Prim Main

key1과 key2를 통해 WDGRAPH와 WDGRAPH[key1]를 반복하면서 WDGRAPH의 값을

graph의 엣지 값으로 추가하고, WDGRAPH에 저장된 weight 값을 graph와, data에 저장한다.
list형 변수 path를 만들고, prev에 prim값을 저장한다. prev를 이중으로 반복하면서 path의 엣지 값을 저장하여 경로를 나타낸다.

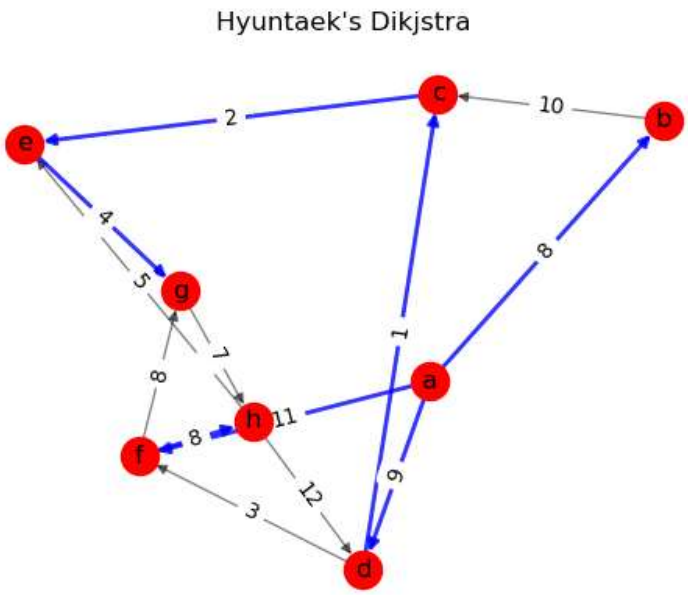
4.결과화면

1) 내가 만든 Dijkstra와 Prim 그래프



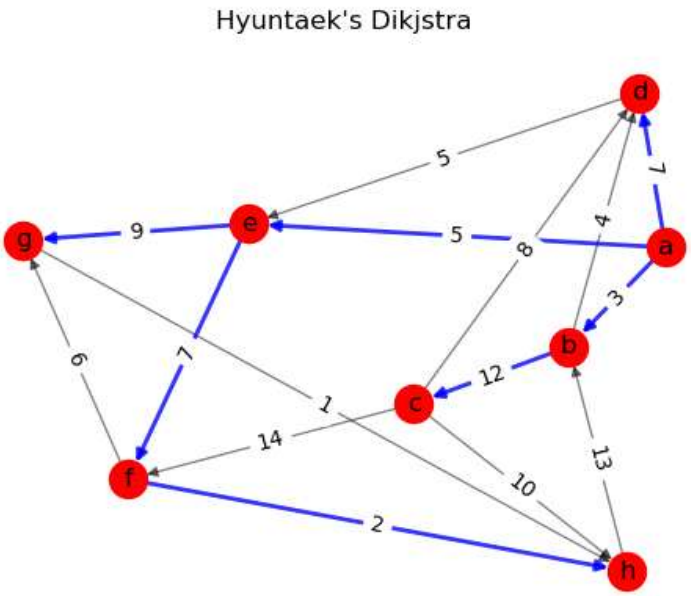
2) 제공 Dijkstra 결과, 그래프

```
Hyuntaek's Dijkstra
({'a': 0, 'b': 8, 'c': 10, 'd': 9, 'e': 12, 'f': 11, 'g': 16, 'h': 19}, {'a': None, 'b': 'a', 'c': 'd', 'd': 'a', 'e': 'c', 'f': 'a', 'g': 'e', 'h': 'f'})
```



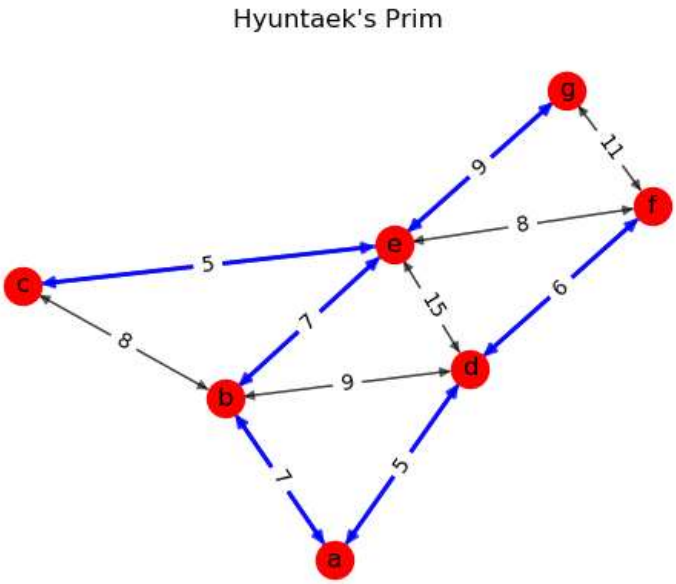
3) 만든 Dijkstra 결과, 그래프

```
Hyuntaek's Dijkstra
({'a': 0, 'b': 3, 'c': 15, 'd': 7, 'e': 5, 'f': 12, 'g': 14, 'h': 14}, {'a': None, 'b': 'a', 'c': 'b', 'd': 'a', 'e': 'a', 'f': 'e', 'g': 'e', 'h': 'f'})
```



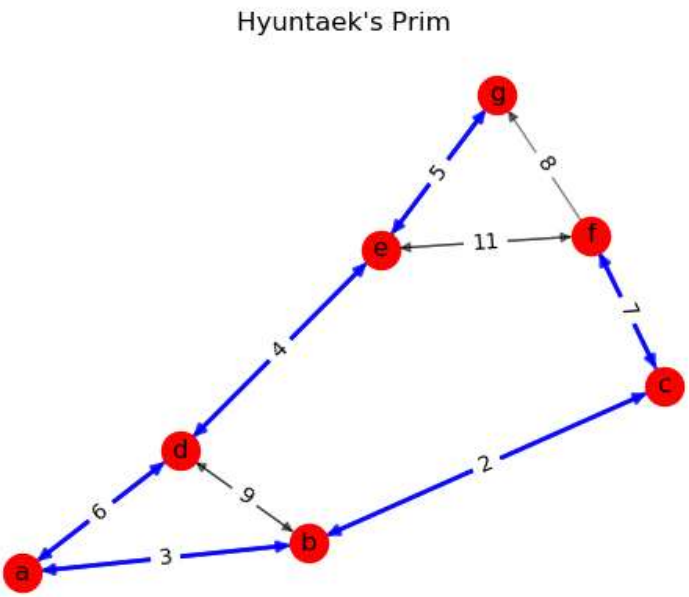
4) 제공 Prim 결과, 그래프

```
Prim 출력
{'a': {'d': 5, 'b': 7}, 'b': {'a': 7, 'e': 7}, 'c': {'e': 5}, 'd': {'a': 5, 'f': 6}, 'e': {'b': 7, 'c': 5, 'g': 9}, 'f': {'d': 6}, 'g': {'e': 9}}
```



5) 만든 Prim 결과, 그래프

```
Prim 출력
{'a': {'b': 3, 'd': 6}, 'b': {'a': 3, 'c': 2}, 'c': {'b': 2, 'f': 7}, 'd': {'a': 6, 'e': 4}, 'e': {'d': 4, 'g': 5}, 'f': {'c': 7}, 'g': {'e': 5}}
```



3 networkx

Python 3에는 Graph를 표현하고 시각화를 도와주는 도구가 있다. `networkx`라는 도구는 지난 시간에 정렬 성능 비교를 위하여 사용했었던 `matplotlib`를 이용하여 Graph를 시각화한다. 예를 들어 fig. 3는 fig. 4처럼 시각화할 수 있다.

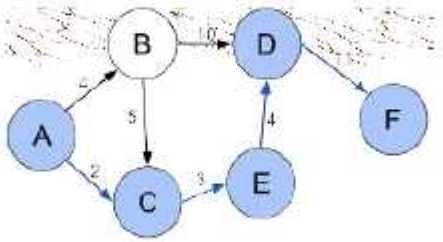


Figure 3: 최단거리 그래프

Figure 4처럼 시각화하기 위하여 `networkx` 라이브러리를 설치해야한다. `pip`를 이용하여 설치한 후 아래 소스코드를 입력하면 같은 그래프가 출력될 것이다. 매 실행시마다 노드의 위치가 바뀌기 때문에 구조는 다를 수 있다. 만약 '무방향 그래프'를 표현해야한다면 `nx.Graph()`로 생성하면 된다.

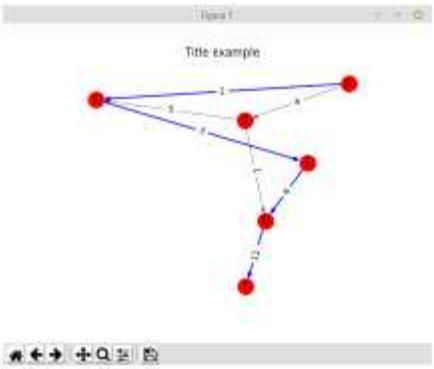


Figure 4: 최단거리 그래프의 networkx를 활용한 시각화

4 최단 거리 알고리즘: Dijkstra

다익스트라(Dijkstra) 알고리즘은 그래프 노드 사이의 최단 거리를 찾는 알고리즘 중 가장 유명한 알고리즘이다. 또한 '알고리즘'이라는 교과목을 이야기할 때 가장 대표되는 알고리즘이기도 하다. 다익스트라 알고리즘은 아래 수도코드로 구현이 된다.

Listing 4: 다익스트라 알고리즘 수도코드

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:                  //
           Initialization
6          dist[v] <- INFINITY                      // Unknown
               distance from source to v
7          prev[v] <- UNDEFINED                     // Previous
               node in optimal path from source
8          add v to Q                               // All nodes
               initially in Q (unvisited nodes)
9
10     dist[source] <- 0                            // Distance
           from source to source
11
12     while Q is not empty:
13         u <- vertex in Q with min dist[u]        // Node with
               the least distance will be selected first
14         remove u from Q
15
16         for each neighbor v of u:                // where v is
               still in Q.
17             alt <- dist[u] + length(u, v)
```

```
18              if alt < dist[v]:                // A shorter
                    path to v has been found
19                  dist[v] <- alt
20                  prev[v] <- u
21
22      return dist[], prev[]
```

Listing 5: dict타입 그래프를 위한 다익스트라 알고리즘 스켈레톤 코드

```
1  import sys
2
3  # FILL_HERE: contents of WDGRAPH02
4  WDGRAPH02 = {
5      }
6
7
8  def dijkstra(wdgraph, start_node):
9      qnode = set()
10     dist = dict()
11     prev = dict()
12
13     # FILL_HERE: initialize variables(multi lines)
14
15     # FILL_HERE: set distance 0 from start node to start
           node
16
17     while len(qnode) != 0:
18         udist = sys.maxsize
19         unode = None
20         for node in qnode:
21             if dist[node] < udist:
22                 udist = dist[node]
23                 unode = node
24         # FILL_HERE: remove node from queue
25
26         # (Most important!) FILL_HERE: compute new
               distance(multi lines)
27
28     return dist, prev
```

5 최소 비용 신장 트리 알고리즘: Prim

모든 노드를 최소한의 비용(에지)로 연결하려면 어떻게 해야할까? 최소 비용 신장 트리 알고리즘(MST)어떤 그래프에 포함된 노드를 모두 연결하기 위한 최소 비용 에지를 찾아내는 알고리즘이다. Prim's algorithm은 MST의 대표 알고리즘으로 컴퓨터 네트워크 노드 연결에서 굉장히 자주 사용된다.

Listing 6: 프림 알고리즘 수도코드

```
1 Initialize a tree with a single vertex, chosen
    arbitrarily from the graph.
2 Grow the tree by one edge: of the edges that connect the
    tree to vertices not yet in the tree, find the minimum
    -weight edge, and transfer it to the tree.
3 Repeat step 2 (until all vertices are in the tree).
```

프림 알고리즘은 굉장히 많은 방법으로 구현될 수 있기 때문에 수도 코드도 문장 형태로 전파되고 있다. 위의 수도 코드를 dict 타입 그래프 대상으로 구현하면 아래의 코드가 된다.

Listing 7: dict타입 그래프를 위한 프림 알고리즘 스켈레톤 코드

```
1  # FILL_HERE: contents of WUGRAPH01
2  WUGRAPH01 = {
3  }
4
5
6  def search_min(wugraph, an, rn):
7      src_node = None
8      dst_node = None
9      dist = float('inf')
10     for src in an:
```

```
11          for dst in wugraph[src]:
12              if dst in rn:
13                  if (wugraph[src])[dst] < dist:
14                      dist = (wugraph[src])[dst]
15                      src_node = src
16                      dst_node = dst
17
18      return src_node, dst_node, dist
19
20
21  def prim(wugraph):
22      mst = dict()
23      added_node = set()
24      remain_node = set()
25
26      for node in wugraph.keys():
27          mst[node] = dict()
28
29      # FILL_HERE: initialize remain_node
30
31      import random
32      start_node = random.choice(list(wugraph.keys()))
33
34      # FILL_HERE: handle start_node - add to added_node,
            remove from remain_node
35
36      # (Most important!) FILL_HERE: compute new distance,
            and handle sets - HINT: while loop
37
38      return mst
```