**2022 겨울학기 연구참여 보고서**

**: Adaptable Priority queue, Priority queue를 이용한 Dijkstra Algorithm 구현 및 Digraph RPP Solution 고안**

2023.01.09~2023.02.18 (6주)

Logistics Lab.

지도교수 김병인

멘토 이승엽

참여자 김진우

**목차**

1. **서론**
2. **연구방법**
   1. Shortest Path Problem
      1. Dijkstra Algorithm
      2. Priority Queue, Adaptable Priority Queue
   2. Rural Postman Problem(RPP)
      1. Rpp On Digraph
3. **연구결과**
   1. Input Data
   2. Dijkstra Algorithm
4. **결론**
5. **후기**
6. **참고문헌**
7. **서론**

본 과제는 한 대의 눈 청소차가 한 구역을 이동하며 눈을 치울 때 이동 시간을 최적화하는 경로를 작성하는 문제이다. 해당 구역에는 눈 치우기를 필수적으로 진행해야 하는 도로와 그렇지 않은 도로가 혼합되어 존재하고, 양방향 통행, 단방향 통행이 가능한 도로가 모두 존재한다.

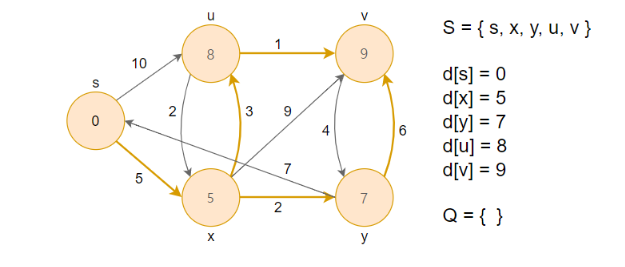
본 문제는 TSP와 달리 Edge를 경유해야 하기 때문에 Arc Routing Problem으로 볼 수 있고, Rural Postman Problem(RPP)으로 해석할 수 있다. RPP는 특정 Graph에서 모든 Edge들을 1번 이상 방문하여 출발 장소로 되돌아오는 최소 비용 문제이고, NP-Complete로 알려져 있다.

RPP를 해결하는 Algorithm은 Undirected Graph에서는 Christofides Algorithm 등이 알려져 있고 최적 성능 대비 <3/2 수준으로 문제를 해결할 수 있음이 알려져 있다.

본 연구에서는 Digraph에서 해당 문제를 해결하기 위한 알고리즘을 고안해보는 것을 목적으로 하고, 문제를 해결하는 과정에서 이동의 최소 비용을 구하는데 사용되는 Shortest Path Problem 해결 알고리즘을 다양한 자료구조를 사용해 해결하는 것을 목적으로 했다.

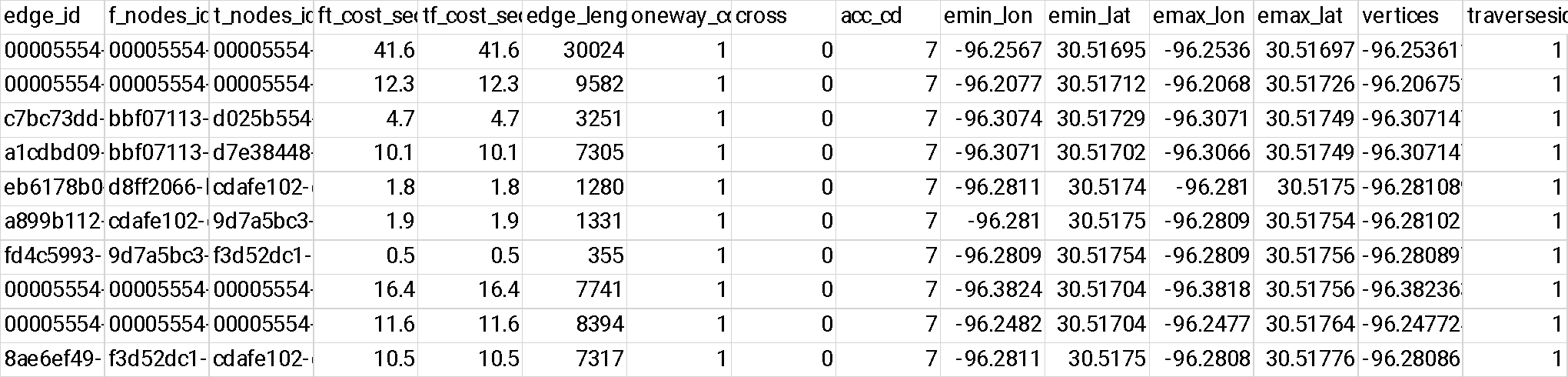
1. **연구방법**
   1. **Shortest Path Problem**

Shortest Path Problem은 주어진 그래프 G의 노드 V들을 잇는 Weighted Edge들의 집합 E가 있을 때, 특정 노드 V1에서 V2로 가는 최소 비용 경로 P = (E1,E2,,,,)를 찾는 문제이다. 각 Edge들의 Weight는 음수일 수도, 양수일 수도 있지만, 본 연구에서 사용하는 Dijkstra Algorithm은 Positive Weighted Edge에 대해서만 유효하기 때문에, 이를 가정할 것이다.



**그림 1 Dijkstra Algorithm 예시**

본 연구에서 사용한 데이터 파일은 다음과 같다.



**그림 2 데이터 파일**

파일 내에는 edge\_id, f\_nodes\_id, t\_nodes\_id, ft\_cost\_sec, tf\_cost\_sec, edge\_length, oneway\_cd, cross, acc\_cd, emin\_lon, emin\_lat, emax\_lon, emax\_lat, vertices, traverseside 등의 parameter가 존재한다. 이 중, Dijkstra Algorithm 구현에 사용한 parameter는 앞의 5가지 parameter들이다. ft\_cost\_sec는 f\_node에서 t\_node로 가는 비용, tf\_cost\_sec는 t\_node에서 f\_node로 가는 비용을 의미한다. edge\_id를 중심으로 연결된 두 노드를 작성했기 때문에, 이를 읽어서 Node\_list로 분리하는 작업이 필요했고, 이를 바탕으로 Graph를 그려야 했다. 본 연구에서는 Unordered Map을 이용해 이를 구현했다.

* + 1. **Dijkstra Algorithm**

Dijkstra Algorithm은 Shortest Path Problem을 해결하기 위해 사용하는 대표적인 알고리즘으로 Single to All 알고리즘이다. 모든 노드를 시작 노드로 설정하면 All to All 알고리즘으로도 활용 가능하다.

Dijkstra Algorithm의 기본적인 실행방식은 다음과 같다.

1.출발 노드를 설정하고, 최단 거리, 방문 여부 테이블을 초기화한다. (출발 노드는 최단거리 0 방문 여부 O로 설정한다. 각 노드에는 Parent Node가 존재한다. 출발 노드에서 reachable한 노드의 경우 모두 출발 노드를 parent node로 설정한다.)

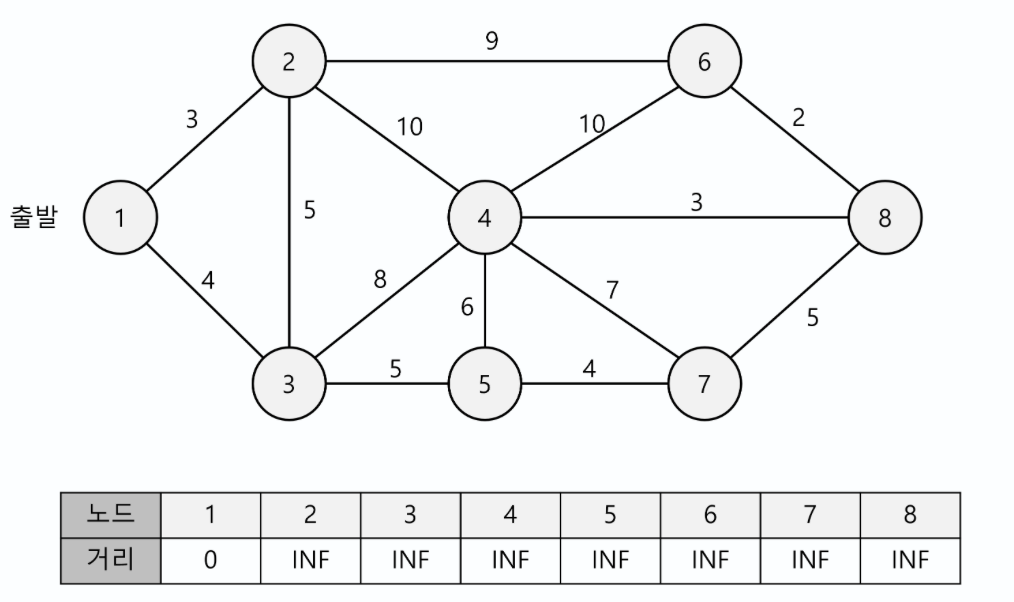
2. 방문하지 않은 노드 중 최단 거리가 가장 짧은 노드를 선택한다.

3. 해당 노드를 거쳐 다른 노드로 가는 비용을 계산하여, 최단 거리 테이블을 업데이트한다. ( , 해당 노드 방문 여부도 o로 표시한다.)

3-1. 만약 라면 parent node를 j로 변경한다.

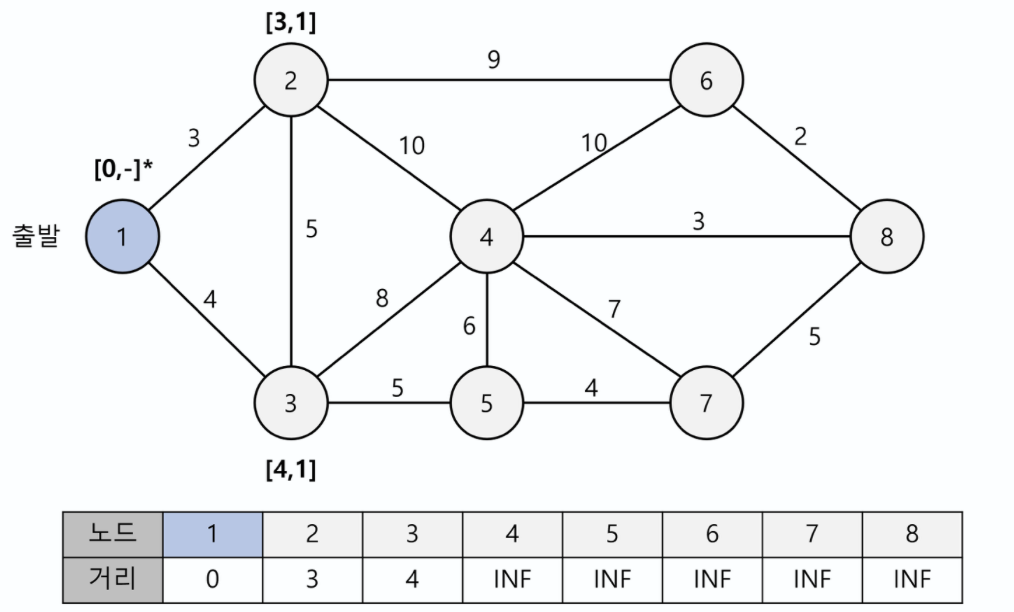
4. 3~4의 과정을 반복한다.

예시를 들어 다시 설명하겠다.



**그림 3 Dijkstra Algorithm 1**

출발 노드를 제외한 모든 노드들에 대해서 거리는 INF 혹은 매우 큰 수로 설정한다. 출발 노드는 거리를 0으로 설정하고, 방문했음을 표시한다.



**그림 4 Dijkstra Algorithm 2**

텍스트, 손목시계이(가) 표시된 사진

자동 생성된 설명

**그림 5 Dijkstra Algorithm 3**

텍스트, 손목시계이(가) 표시된 사진

자동 생성된 설명

**그림 6 Dijkstra Algorithm 4**

텍스트, 손목시계, 시계이(가) 표시된 사진

자동 생성된 설명

**그림 7 Dijkstra Algorithm 5**

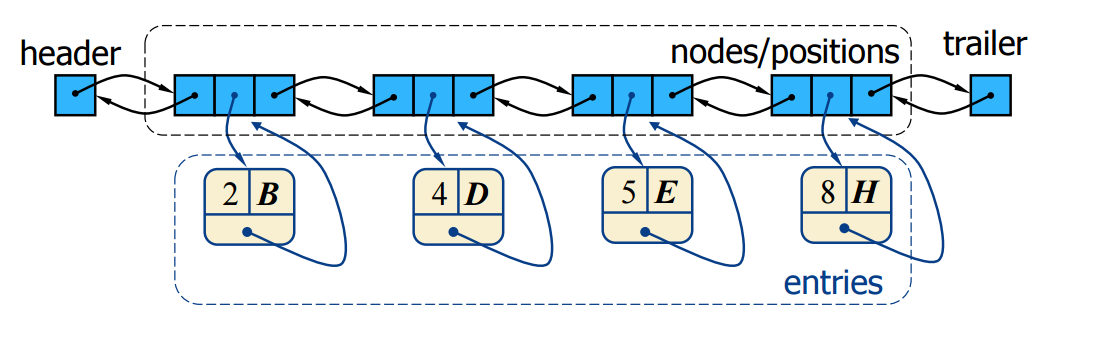
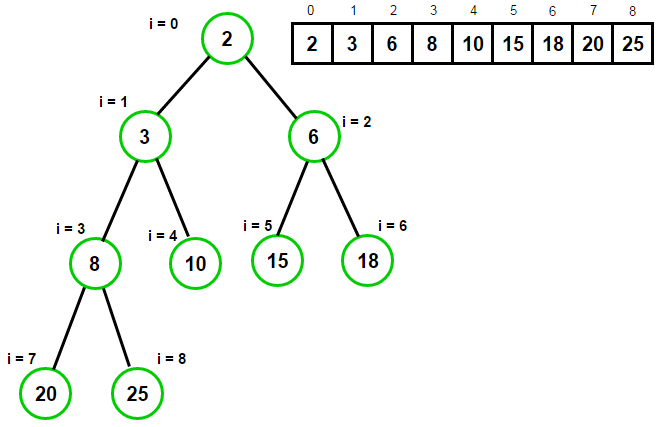
그림 4,5,6처럼 거리 및 방문 여부 업데이트를 계속 진행하면 그림 5의 결과를 얻을 수 있다.

* + 1. **Priority Queue, Adaptable Priority Queue**

잘 알려진 Queue 자료구조는 FIFO(First In First Out)방식으로 데이터를 처리하는 자료구조형이다.

Priority Queue(우선순위 큐)는 데이터에 우선순위를 정해두는 Queue구조라고 생각할 수 있다. 여기서 데이터의 우선순위는 적용하기 나름인데, 대표적으로는 Min Priority Queue, Max Priority Queue가 있다. 본 연구에서는 Min Priority Queue를 사용했다. C++ 환경에서는 ‘#include <queue>’, ‘std::priority\_queue’ 구문을 통해 쉽게 구현할 수 있었다.

다음으로 Adaptable Priority Queue에 대해 알아볼 것이다. Adaptable Priority Queue와 Priority Queue의 가장 큰 차이점은 자료구조 내에 있는 데이터에 대한 접근 여부이다. Priority Queue의 경우 이미 데이터 구조 내에 들어간 데이터들에 대해서 순서를 바꾸는 등의 행위를 할 수 없지만, Adaptable Priority Queue의 경우 ‘Update’ 함수를 이용하면 데이터에 접근해 순서를 바꿀 수도, 값을 업데이트할 수도 있다. 이 과정은 기본적인 Enqueue, Dequeue에 비해 cost-expensive한 연산이기 때문에, 값에 접근해야 하는 경우가 많을 때 유용하지만, 그렇지 않다면 오히려 낭비일 수도 있다.



**그림 8 Priority Queue(위), Adaptable Priority Queue(아래)**

Dijkstra Algorithm에서 Adaptable Priority Queue 자료구조를 사용했을 때의 이점은 알고리즘의 효율성이 올라간다는 것이다.

* 1. **Rural Postman Problem**

Rural Postman Problem은 Graph에 포함된 모든 edge를 한 번 이상 거쳐 다시 돌아오는 최소비용 경로를 찾는 문제이다. [[1]](#endnote-1)

RPP를 해결하는 알고리즘으로는 Christofides et al. Algorithm이 알려져 있다. 이 알고리즘은 undirected graph에 적용되는 알고리즘으로 크게 3개의 Phase로 나눠져 있고 그 내용은 아래와 같다.[[2]](#footnote-1)

Phase 1 Graph Transformation

Step 1. Shortest Path Algorithm을 이용해 Artificial Edge를 추가하고 그래프를 Complete Graph로 만들어준다.

Step 2. Complete Graph에서 edge(i,j)의 cost Cij = Cik + Ckj일 경우와 두 edge가 평행하며 비용도 같은 경우 edge를 삭제하는 작업을 추가로 진행한다.

Phase 2 Minimal Spanning Tree

Step 1. GR의 Component들을 이어주는 작업을 한다. 이 때 component를 이어주는 edge는 d(i,j) = d(Ci,Cj) = min(xy){d(x,y) – u(x) – u(y)} (u(x),u(y)는 노드의 degree를 의미한다. )로 정의한다. 이때 edge(x,y)는 G(RC)에 속한다.

Step2. MST Algorithm을 적용해 MST를 얻는다.

Phase 3

Step 1. Minimal Cost Matching Algorithm을 통해 Minimal Cost Matching을 얻을 수 있고, 여기서 나온 오일러 그래프가 문제의 해답이다.

해당 오일러 그래프는 최적 성능 대비 < 1.5 임이 알려져 있다.

해당 알고리즘은 Undirected Graph에 특화되어 있음으로 Digraph에 적용하기 위해서는 변화가 필요하다. 고안한 방법은 아래와 같다. 우선 기본적인 Phase 3단계는 본래 알고리즘과 달라진 것이 없다. 다만 Phase2에서 MST를 구하는 것에 대해서, MST는 Undirected Graph에 적용되는 개념이므로, 이에 상응하는 다른 개념이 필요한데, 이를 Digraph Minimum Spanning Tree(DMST)라고 정의한다. 이 때, DMST를 정의하기 위해서는 2가지 사실을 만족 해야 한다. (여기서 루트 r은 시작 노드를 의미한다.)

Fact 1. DMST는 무방향 그래프로 봤을 때 사이클이 존재하지 않는다. R을 제외한 모든 노드의 indegree가 1이다.

Fact2. 루트가 아닌 모든 정점에 대해서, 해당 정점으로 들어오는 minimum-cost edge를 e(v)라고 하고 그 cost를 w(e(v))라고 할 때, DMST의 cost 총 합 이다. (이는 Fact 1에 의해 자명한 사실이다.)

즉, Phase2에서 Cyclic Component를 찾아서 이 Component들을 서로 연결해준 뒤 루트를 제외한 모든 Node들의 indegree가 1이 되도록 조정해줄 수 만 있다면 DMST를 구할 수 있는 것이다. 다음은 고안한 알고리즘의 Pseudo Code이다.

Phase 1 Graph Transformation

Step 1. Shortest Path 알고리즘을 이용해 Complete Graph를 만들어준다. 이 때 만약 도달할 수 없다면 minimum-cost를 매우 크게 하여 artificial edge를 추가할 수 있다.

Step 2. 기존 알고리즘과 동일한 방식으로 arc를 삭제하되, parallel의 경우 방향까지 동일한 경우에만 삭제하도록 한다.

Phase 2 Directed Minimum Spanning Tree

Step 1. 루트를 제외한 모든 노드에 대해 minimum-cost-in-direction edge를 선택해 Cyclic Component를 찾는다.

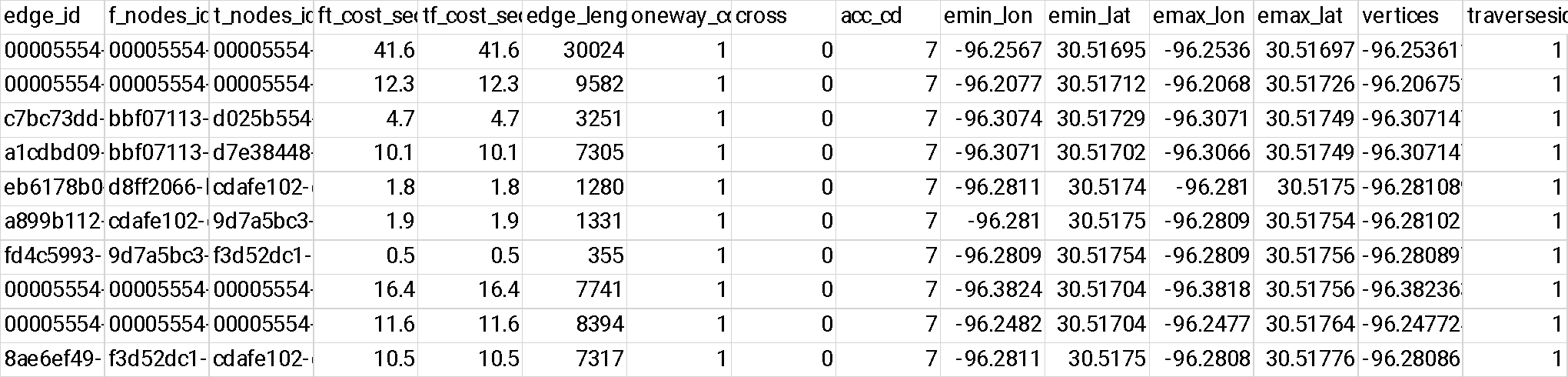
Step 2. 루트로부터 minimum-cost에 있는 node를 선택해 acyclic component를 만들고 해당 acyclic component에서 minimum-cost인 cyclic component를 잇는다. 이때 cycle을 삭제하기 위해 edge를 삭제하는 과정을 반복한다. 이때 edge의 cost w(e) = w(e) – w(e’)을 사용한다. e’은 삭제된 edge를 의미한다. 이렇게 하면 DMST를 얻을 수 있다.

Phase 3. Minimum Cost Matching

Step 1 떨어져 있는 Object들을 Minimum Cost Matching 시켜준다. 이 때 사이클이 생기지 않도록 매칭해야 한다.

Phase 3에 대한 추가적인 고안이 필요하고 위 알고리즘이 optimize된 솔루션을 제공할 지에 대해서는 추가적인 연구와 실험이 필요하다. 본 연구에서는 고안 까지만 진행을 하고 실험을 진행하지 않았기 때문에, 연구 결과에서 추가적으로 다루지는 않도록 하겠다.

1. **연구결과**
   1. **Input Data**



**그림 9 edges.txt 데이터**

위 그림은 본 연구에서 사용한 input data이다. 별도의 예시 데이터 없이 해당 데이터를 사용해 진행을 했다.

* 1. **Dijkstra Algorithm**



**그림 10 Dijkstra with adaptable priority queue 결과**



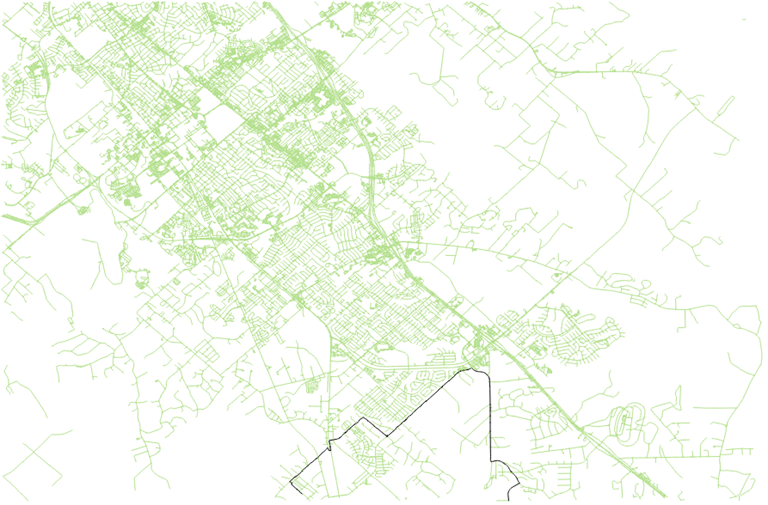
**그림 11 Dijkstra with priority queue 결과**

|  |  |  |
| --- | --- | --- |
|  | Adaptable Priority Queue | Priority Queue |
| # of Nodes | 19958 | 19958 |
| # of Edges | 25495 | 25495 |
| Elapsed time | 125,913ms | 126,175ms |

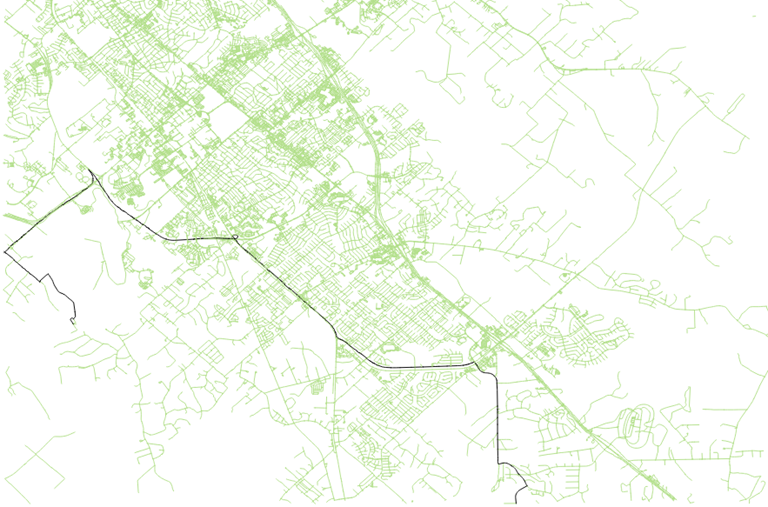
**표 1 Adaptable Priority Queue, Priority Queue 비교**

위 두 output file은 각 데이터 구조를 활용해 Dijkstra Algorithm을 실행시킨 결과이다. 앞 열에서부터, 시작 노드, 도착 노드, 최단 거리, 경로를 나타낸다. 최단거리가 1000000인 경우는 도달할 수 없는 노드를 의미하고, Path에서 시작점과 도착점은 나타내지 않았다.

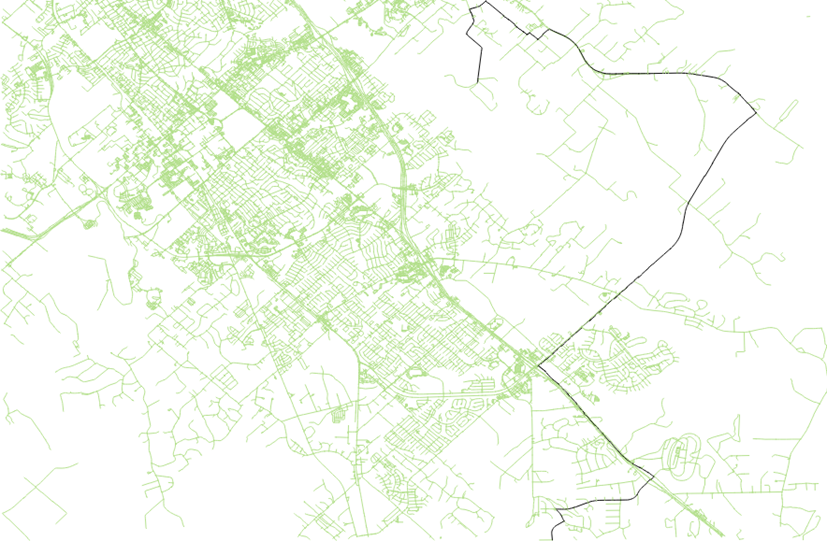
연구 결과 Adaptable Priority Queue를 사용했을 때 근소한 차이지만, 시간이 더 적게 걸리는 것을 확인할 수 있었다. 10회 실험을 진행했는데, 모두 Adaptable Priority Queue를 사용했을 때 시간이 더 적게 걸렸다.



**그림 12 Node0 -> Node 4 시각화**



**그림 13 Node 0 -> Node 3473 시각화**



**그림 14 Node 0 -> Node 15650 시각화**

위 3가지 그림은 Output file의 path 데이터를 바탕으로 시각화 시켜 나타낸 그림이다.

m개의 edge, n개의 node가 있다고 할 때, Adaptable Priority Queue를 사용했을 때 시간 복잡도는 O(mlogn), Priority Queue를 사용했을 때도 O(mlogn)이라고 알려져 있지만, 이는 Queue에 있는 데이터들을 효율적으로 update할 수 있을 때의 가능한 것이고, 일반적으로는 Priority Queue를 사용했을 때 O(mlogn)\*O(n) = O(mnlogn) (이때, O(n)은 데이터를 찾고 지운 뒤 다시 넣는 과정에서 발생하는 시간 복잡도이다.)

1. **결론**

Dijkstra 알고리즘에 대해 두 가지 자료 구조형으로 실행을 했을 때, 총 19958개의 Node, 25495개의 Edge가 있을 때 All to All 120~130초 사이의 시간에 문제를 해결할 수 있었다. 실제 프로그램을 모두 돌려보면, 알고리즘을 돌리는 시간은 매우 적지만, output file을 출력하는 시간이 길어 running time이 길어 졌음을 알 수 있다.

본 연구 과정에서는 adaptable priority queue 모듈을 만들어 해당 모듈을 사용하였는데, ‘정보시스템기술’ 수업을 미루어 볼 때 모듈을 사용하게 되면 모듈을 import하고 사용하는 시간이 들어 시간 효율성이 떨어진 경험이 있었다. 이번 연구에서도 adaptable priority queue를 사용했을 때 시간적으로 많은 효율을 취하지 못 했는데 비슷한 이유일 것이다. Priority queue stl을 사용해 구현을 할 수만 있다면 더 시간이 줄어들 것이다. 이 부분에 대해서는 검증을 위한 실험이 필요하다.

Digraph RPP solution 도출에 대한 고안만 진행하고 실험을 진행해보지 못 한데에 대한 아쉬움이 남는다. RPP의 경우 Undirected Graph에서는 해결 알고리즘이 있지만, Digraph에서의 해결은 제대로 나와있지 않은 것 같다. 고안한 방법을 토대로, 실험을 진행해 실현가능성에 대해 논하고 알고리즘이 개발되는 것을 기대한다.

1. **후기**

대학에 들어와서 가장 흥미롭게 들은 두 과목을 고르라면, 주저없이 OR1, 정보시스템기술을 선택할 것 같습니다. 최적화 문제를 이해하고 풀어내는 것도 흥미로웠지만, 정보시스템기술에서 했던 것처럼, 알고리즘을 작성해 컴퓨터로 해결하는 과정이 저에게는 더 흥미롭게 느껴져 이번 겨울학기 첫 연구참여를 김병인 교수님 연구실에서 진행하게 되었습니다.

저에게 있어서 이번 학기 연구참여의 목적은 대학원 진학에 대한 확신 갖기 였습니다. 생각했던 대학원 생활과 무엇이 다른 지, 어떻게 다른 지 몸소 체험 해보고 싶었습니다. 연구참여를 진행하면서 다시 공부하는 C++, 처음 제대로 읽어보는 원문 논문들이 저를 두렵게 만들기도 했지만, 시간을 들여 내용을 곱씹어 나가는 과정이 생각보다 더 흥미로웠던 것 같습니다.

처음 연구참여에 임했을 때는 일정도 변동이 없었고, 방학 계획을 세운 것처럼 잘 흘러가는 듯 싶었으나, 새내기새로배움터를 진행하며 일정이 너무 유동적으로 변하고, 타이트한 마감 일정 등으로 연구참여에 제대로 힘을 쏟지 못한 것 같아 많이 아쉬움이 남습니다. 연구참여 초기에 설정한 목표인 Digraph RPP해결이라는 목표를 달성하지 못한 것이 아쉬움으로 남고, 가능하다면 다시 도전해보고 싶다는 생각도 듭니다.

이번 연구 참여를 진행할 때 저에게 기회를 주시고, 관심을 가져 주시며 피드백 해 주신 김병인 교수님께 감사드리고, 바쁘신 와중에도 매주 저와 개인 미팅을 가져 주시며 가르쳐 주시고, 데이터를 시각화 해 주신 이승엽 사수님께 감사의 말씀을 드립니다.

1. **참고문헌**

1. Pearn, W. L., & Wu, T. C. (1995). Algorithms for the rural postman problem. Computers & Operations Research, 22(8), 819-828. [↑](#endnote-ref-1)
2. Pearn, W. L., & Wu, T. C. (1995). Algorithms for the rural postman problem. Computers & Operations Research, 22(8), 819-828. [↑](#footnote-ref-1)