5주차 과제 / 그리디 알고리즘 - 최소신장트리 찾기

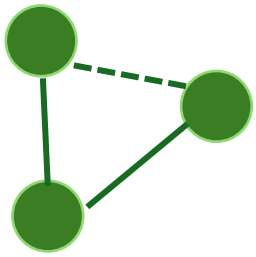
최소신장트리(MTS) - 사전 공부(기억이 안 나서…. 이거 공부부터….)

신장트리: 모든 정점을 포함, 싸이클이 없는 트리

최소신장트리: 가중치들의 총합이 최소가 되는 신장트리

-> 노드가 n개일 때, 간선의 수는 n-1개가 된다.

: 직관적으로 이해하는 사람도 있겠지만, 잘 모르겠어서 이것저것 찾아봤는데 그냥 예시를 들면 편하다.



: 그리기 힘들어서 세개만 했는데 간선의 개수가 3개(n개)가 되는 순간에 순환이 생긴다. 뭐 하나를 빼서 2개(n-1개)로 맞춰줘야 순환이 안 생긴다. 5개짜리 손으로 해봤는데 진짜 그랬다.

: 사실 어떻게 증명(?) 하는지는 모르겠다. 이산수학때도 그냥 외웠던 것 같으므로 외우자.

1학년 때 배운 것들이니만큼 이번에는 혼자 짜보고 싶어서 최대한 노력해 보았다.

작년의 C 코드가 남아 있어서 같은 로직을 사용하여 구현했다.

<u>크루스칼 알고리즘</u>

1. 모든 간선들의 가중치를 기준으로 오름차순으로 정렬

2. 가장 작은 가중치를 가지는 간선을 선택 -> 사이클 판단

3. 종료시점까지 반복

종료 조건: 간선의 수가 N-1개가 되는 순간에 종료

(오…. 기억난다.)

중요한 건 사이클을 판단해주는 유니온 어쩌고 함수였다. 같은 부모 노드를 가지는(출발 노드가 똑같은) 정점들은 버리는 함수이다.

아래는 작년에 짠 C 코드이다.

```
int vFind(int vertices[], int vNum) {
	if (vertices[vNum] == -1) {
		return vNum;
	}
	while (vertices[vNum] != -1) {
		vNum=vertices[vNum]; // 같은 집합에 있는 정점으로 가라. 긍까 b(1)이 -1 c(2)가 1, g가 2인데 내가 g을 보냈으면 g->c->b로 이동해서 -1을 찾음

		return vNum; // 그때 vNum 돌려줌
	}
}

void vUnion(int vertices[], int vNum1, int vNum2) {
	vertices[vNum2] = vNum1;
}

void kruskal(GraphType* G, Edge* edges[], int vertices[]) {
	int eCnt = 0; int i = 0;

	int vNum1, vNum2;
	Edge* p;

	while (eCnt < G->vCount - 1) {
		p = edges[i];
		vNum1 = vFind(vertices, p->v1 - 97); // 캐릭터를 인트로 해주려고 -97
		vNum2 = vFind(vertices, p->v2 - 97);

		if (vNum1 != vNum2) {
			eCnt++;
			printf("%d. [%c%c %d]₩n", eCnt, p->v1, p->v2, p->weight);
			vUnion(vertices, vNum1, vNum2);
		}
		i++;
	}
}
```

크루스칼 함수만 보면 우선 그래프와 정점 배열들, 가중치 정보를 가지고 있는 배열을 받는다.

eCnt로 간선들의 개수를 세어줄 거고, i를 사용해서 가중치 배열을 인덱싱한다.

와일문으로 들어가서 eCnt가 N-1이 될 때까지 돌린다.

- vFind로 두개의 간선의 인덱스를(노드 번호겠지…?) 찾는다(선택되지 않은 인덱스: 이때 선택 안 된 애들을 -1로 해놨던 기억이 있다)

- 두 인덱스가 다르다면 eCnt를 증가시킨 후 유니온을 사용해 하나로 묶어준다 -> 순환을 방지

- i 증가

이해하는데 한참 걸렸지만 대강 어떤 로직인지는 알 것 같아서 파이썬으로 코딩을 시도했다.

아, 하려고 했는데 그래프 만드는 법을 몰라서 C에서 했던 대로 하려다가 완전히 망했다. 그리고 챗지피티가 어떻게 만드는지 알려줬다

```
class Edge:
    def __init__(self, v1, v2, weight):
        self.v1 = v1
        self.v2 = v2
        self.weight = weight

class GraphType:
    def __init__(self, vCount):
        self.vCount = vCount
        self.edges = []

    def add_edge(self, v1, v2, weight):
        self.edges.append(Edge(v1, v2, weight))
```

자바처럼 클래스를 만들어서 객체를 만들 수 있게 해주는데…. 사실 자바 기억이 안 나서 C의 struct 같은 거라고 이해했다. 아무튼.

엣지 클래스에서 정점을 생성한다. Init는 수업시간에 배웠고 self 같은 경우 자바의 this 와 아주 비슷하다. 나를 초기화하는 것이다.

그래프타입 클래스에서 vCount(정점 개수)와 가중치 배열을 담은 edge 배열을 만들어주고, 클래스 내에 edge 를 추가할 수 있는 add_edge 함수를 선언한다.

레포트 잘 써보고 싶었는데 순서가 꼬였다 어쨌든…. 내가 쓴 크루스칼은 아래와 같다.

```
def vFind(vertices, vNum):
    if vertices[vNum] == -1:
        return vNum # 아직 선택이 안 된 정점이라면 그대로 리턴
    while vertices[vNum] != -1:
        vNum = vertices[vNum]
        return vNum
        # 아니라면 -1 이 나올 때까지 같은 집합 내에서 이동.
        # b(1)이 -1 c(2)가 1, g(?)가 2 인데 내가 g 를 보냈으면 g->c->b 로 이동해서 -1 을 찾음

def vUnion(vertices, vNum1, vNum2):
    vertices[vNum2] = vNum1
    # 하나의 집합으로 묶어주고 같은 집합 내 다른 정점 정보를 담은 숫자를 업데이트

def kruskal(G, vertices):
    eCnt = 0
    i = 0
    G.edges.sort(key=lambda edge: edge.weight)  # 가중치 기준, 오름차순으로 정렬

    while eCnt < G.vCount - 1:
        p = G.edges[i]
```

```
        vNum1 = vFind(vertices, p.v1)  # 이번에는 정점 번호 숫자로 할거라 -97 안한다.
        vNum2 = vFind(vertices, p.v2)

        if vNum1 != vNum2:
            eCnt += 1
            print("선택된 정점의 수: ", eCnt, "[", "정점 1:", p.v1, "정점 2: ", p.v2, "가중치: ", p.weight, "]") # 잘 되고 있는지 찍어보기
            vUnion(vertices, vNum1, vNum2)
        i += 1

vertices = [-1] * 7  # 정점 처음에 -1로 설정(전체 미방문 처리)
G = GraphType(7)  # 그래프 생성


# 간선 추가는 귀찮아서 챗지피티 시켰다
edges_info = [
    (0, 1, 10), (0, 6, 3), (0, 5, 7),
    (1, 3, 9), (1, 6, 6),
    (2, 3, 2), (2, 6, 4),
    (3, 6, 10), (3, 4, 4), (3, 5, 11),
    (4, 5, 5),
    (5, 6, 2)
]

for v1, v2, weight in edges_info:
    G.add_edge(v1, v2, weight)

kruskal_result = kruskal(G, vertices)
```

결과(순서대로 파이썬, c)

```
"C:\computer algorithm\pythonProject\venv\Scripts\p
선택된 정점의 수:  1 [ 정점 1: 2 정점 2:  3 가중치:  2 ]
선택된 정점의 수:  2 [ 정점 1: 5 정점 2:  6 가중치:  2 ]
선택된 정점의 수:  3 [ 정점 1: 0 정점 2:  6 가중치:  3 ]
선택된 정점의 수:  4 [ 정점 1: 2 정점 2:  6 가중치:  4 ]
선택된 정점의 수:  5 [ 정점 1: 3 정점 2:  4 가중치:  4 ]
선택된 정점의 수:  6 [ 정점 1: 1 정점 2:  6 가중치:  6 ]

Process finished with exit code 0
```

```
1. [cd 2]
2. [fg 2]
3. [ag 3]
4. [cg 4]
5. [de 4]
6. [bg 6]
```

* c에서는 알파벳으로 했어서 정점 이름이 다르긴 한데 a 부터 0번 인덱스로 생각하면 결과는 일치한다.

* 정점 7개 -> 간선이 6개가 되었을 때 종료되고 있다

시간복잡도

while문 도는 거…. 밖에 없는 것 같아서 O(1)인가? 싶어서 검색했는데 아니었다. 아래는 티스토리 복붙 글이다.

정점의 개수 **V**, 간선의 개수 **E** 일때, 각 단계의 유니온파인드 과정에서 소요되는 시간을 **O(V)**로 러프하게 잡아도 모든 간선을 정렬하는 과정에서 이미 **O(ElogE)**의 시간이 소요되기 때문에 전체 시간복잡도는 **O(ElogE)**로 표현될 수 있다.

또한, 만약 그래프가 Dense 하다면, **E = V^2** 이므로, **O(ElogV)**로도 나타낼 수 있다.

마지막 줄이 무슨 말인가 싶어서 챗지피티한테 물어봐서 이해했다.

그래프가 dense(밀집)하다는 것은 그래프의 정점들 사이에 가능한 많은 간선이 존재한다는 것을 의미합니다. 이는 정점 사이에 연결되지 않은 경로가 거의 없다는 것을 뜻하며, 그래프의 모든 정점 쌍 사이에 간선이 있을 경우, 그래프는 완전 그래프(complete graph)가 됩니다.

-> 이 뒤에 말이 조금 길어서 잘랐는데, 간선의 합이 vC2(조합)이 되어서 v(v-1)/2로 나타내게 된다고 한다.

아무튼 가장 많은 시간을 쓰는 건 처음에 가중치 오름차순 정렬이라고 하고, sorted 이기 때문에 O(nlogn)의 시간복잡도를 가지며, 정점의 개수 E가 n에 대응된다. 그러니까 O(ElogE)

그리고 최선의 경우, 즉 정렬할 vertex가 적어서 그래프가 스펄스할 경우 O(ElogV)로 간주될 수도 있다.

성능 테스트

프림이랑 비교하는 것도 할 건데, 단독 성능도 궁금해서 해봤다. Dense할 경우와 sparse 할 경우를 나눠서 측정해 보려고 한다. 덴스, 스펄스한 그래프 생성하는 함수는 깃허브에서 주웠다.

음….

```
IndexError: list index out of range
```

챗지피티는 vFind와 while문을 수정하기를 제안했다.

`vFind` 함수 수정:

`vFind` 함수에서 재귀 호출을 올바르게 처리하도록 수정하겠습니다. 불필요한 `while` 루프를 제거하고, 대신 재귀적으로 부모 정점을 찾도록 합니다.

python Copy code

```python
def vFind(vertices, vNum):
    if vertices[vNum] == -1:
        return vNum
    return vFind(vertices, vertices[vNum])
```

`kruskal` 함수 내에서 `while` 루프가 `G.edges` 리스트의 범위 내에서 실행되도록 조건을 추가합니다.

python Copy code

```python
def kruskal(G, vertices):
    eCnt = 0
    i = 0
    G.edges.sort(key=lambda edge: edge.weight)

    while eCnt < G.vCount - 1 and i < len(G.edges):
        p = G.edges[i]
        vNum1 = vFind(vertices, p.v1)
        vNum2 = vFind(vertices, p.v2)

        if vNum1 != vNum2:
            eCnt += 1
            vUnion(vertices, vNum1, vNum2)
        i += 1
```

아래 while은 이해했는데 vFind는 뭐가 문제인지 모르겠어서…. 또 물어봤다.

또는 `while` 루프를 사용해도 되지만, 이 경우에는 `return` 문을 루프 밖으로 옮겨야 합니다:

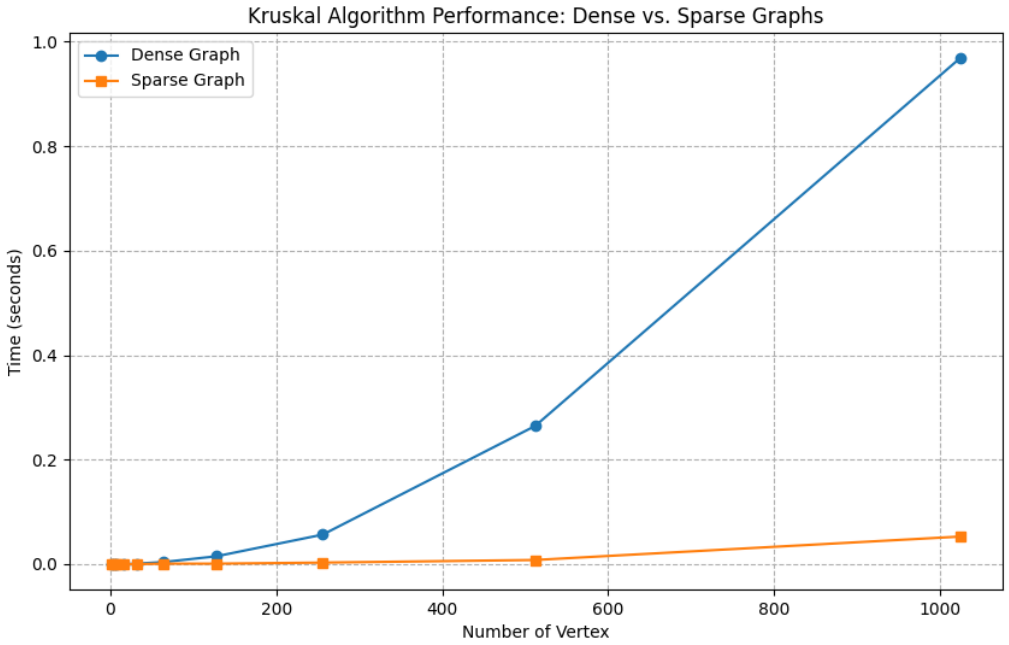
python Copy code

```python
def vFind(vertices, vNum):
    while vertices[vNum] != -1:
        vNum = vertices[vNum]
    return vNum
```

아니 C 수업 받을때 졸면서 코드를 짰나보다. 리턴문을 와일 안에 넣어둬서 바로 리턴을 시키고 있

었다. 반복도 안하고 바로 리턴해버리면…. 이상한게 당연하다. 근데 테스트에서는 왜 잘 돌았지? 아무튼 발견해서 다행이다.

본론으로 돌아가서 성능테스트 그래프를 첨부한다. 2의 거듭제곱으로 표현했다.



처음에는 둘 다 비슷하다가, 정점의 개수가 많아지면서부터 가파르게 증가한다. 증가율은 dense할 경우 훨씬 크다.

프림 알고리즘

1. 정점 하나 선택, 인접한 정점 중 가장 작은 가중치를 가지는 정점을 선택

2. 선택된 모든 정점과 인접한 간선 중 싸이클을 만들지 않으면서, 가장 작은 가중치를 가지는 애를 선택

3. 선택된 간선의 수가 n-1이 될 때까지 반복

작년에 C로 프림 알고리즘을 짜면서 연결리스트와 우선순위큐를 사용했던걸로 기억한다. 아무튼 아래는 작년의 C 코드이다

```
void prim(GraphType* G, element vName, int dist[]) {
	Vertex* v = findVertex(G, vName);
	AdjacentVertex* a = NULL;
	element name;

	dist[vName - 97] = 0; // 시작 정점을 0으로

	for (int i = 0; i < G->vCount; i++) {
		name = getMinVertex(G, dist);
		v = findVertex(G, name);
		v->isVisit = TRUE;
		printf("[%c] ", v->vName);

		for (a = v->aHead; a != NULL; a = a->next) { // 방문 정점의 인접정점 구경
			v = findVertex(G, a->aName); // 인접정점의 인접정점을 구경
			if (v->isVisit == FALSE && a->e->weight<dist[a->aName-97]) {
				dist[a->aName - 97] = a->e->weight;
			}
		}
	}
}
```

다 가져오면 코드가 너무 길어져서 프림만 가지고 왔다.

우선 그래프와 시작 정점의 이름, 거리 배열(inf로 초기화되어 있는)을 보낸다

함수 내에서 날아온 시작 정점의 이름과 같은 이름의 정점을 G에서 찾아 v에 넣는다

인접정점의 이름, 간선을 가르키는 포인터, 다음 인접 정점을 가르키는 포인터가 들어 있는 AdjacentVertax 구조체의 포인터를 선언하고 초기화한다.

모두 inf로 초기화되어 있는 dist 배열을 0으로 초기화한다

최소의 가중치를 가지는 간선의 정점(방문하지 않은)을 리턴하는 함수에서 받아온 인접 정점의 이름을 변수에 넣는다

V에 간선을 찾아 넣고, 방문 기록을 남긴다.

이너루프를 돌면서 현재 방문한 정점의 인접 정점을 구경한다

인접정점과 인접한 정점을 구경하면서, 방문 기록이 없고, 새로운 최소 경로를 찾으면 업데이트한다.

(와 어렵다….)

비슷한 로직을 사용해서 파이썬 코드를 짜 보았다. 포인터 대신 인스턴스 생성한 거 빼고는 거의 다 똑같았다. 그리고 이너루프를 for 대신 while을 써서 구현했다 특별한 이유는 없고, 너무 그대로 쓰는거 같았기 때문이다.

```
def prim(G, vName, dist):
    v = findVertex(G, vName)
    dist[vName] = 0

    for _ in range(G.vCount): # 정점 수만큼 돌리기
        name = getMinVertex(G, dist)
        if name is None:  # 모든 정점이 방문된 상태
            break # 포문을 끝내라
        v = findVertex(G, name)
        v.isVisit = True
        print("선택된 정점:", v.vName)

        a = v.aHead
        while a is not None:
            v = findVertex(G, a.aName)
            if not v.isVisit and a.weight < dist[a.aName]:
                dist[a.aName] = a.weight
            a = a.next
```

코드가 너무 길어져서 findVertex같은 애들은 생략했지만, 열심히 구현했다. 사실 챗지피티의 도움을 많이 받았다…. 혼자 해보고 싶었는데….

결과(순서대로 파이썬, c)

```
선택된 정점: 0
선택된 정점: 6
선택된 정점: 5
선택된 정점: 2
선택된 정점: 3
선택된 정점: 4
선택된 정점: 1
```

```
[a]
[g]
[f]
[c]
[d]
[e]
[b]
```

이것 또한 똑같이 잘 나온다.

* 티스토리에서 주워본 프림 알고리즘

아무리 봐도…. 저렇게 하는게 아닌 것 같아서 좀 찾아봤다. 이게 C언어도 아니고 조금 더 고등한(?) 방법이 있어 보이는데, 혼자서는 못하겠어서 주웠다.

https://c4u-rdav.tistory.com/49

읽어보니까 확실하게 기억이 났다. 정점 하나를 집어넣고 인접정점들을 모두 큐에 넣은 후, 처음 집어넣었던 정점을 빼서 방문처리한다. 또 큐에서 하나를 빼서 인접정점들을 집어넣고 어쩌고저쩌고를 반복해서 큐가 빌 때까지 돌리는 거였다. 우선순위큐로 구현한 c 코드를 도저히 못 찾겠어서, 나중에 시간 나면 한번 해보기로 했다.

-> 성능 차이가 있어서, 원하는 결과를 얻기 위해 티스토리의 코드를 사용했다. 다음에는 나도 꼭 고등한 방식으로 해보고 싶다. 어렵다…. ㅜㅜ

시간복잡도

처음에 내가 구현할 때, 이거 루프 두개 돌리면서 다 방문해 봐야 하는거니까 무조건 O(n^2)일 것이라고 생각했고 맞았다.(정점 개수가 v일 때 O(v^2))

그런데 내가 성능 측정을 해본 건 힙으로 구현한 것이다. 그래프를 먼저 그려본 후 추측하기로, 모양이 크루스칼과 비슷하니 O(ElogE)이겠다 라고 생각했다. 하지만 좀 알아보니까 O(ElogV)였다. 둘에 대한 비교는 뒤에 계속한다.

프림만 놓고 생각해보자면, 해당 알고리즘은 우선순위큐(힙)를 사용해서 구현되어 있다. 알고리즘에서 가장 크게 영향을 미치는 부분은 우선순위큐 삽입, 삭제와 관련되어 있고, 이는 O(log N)의 시간복잡도를 가진다. 또한 최악의 경우 모든 간선 E에 대해 연산을 해야 하므로 O(ElogV)가 완성된다.

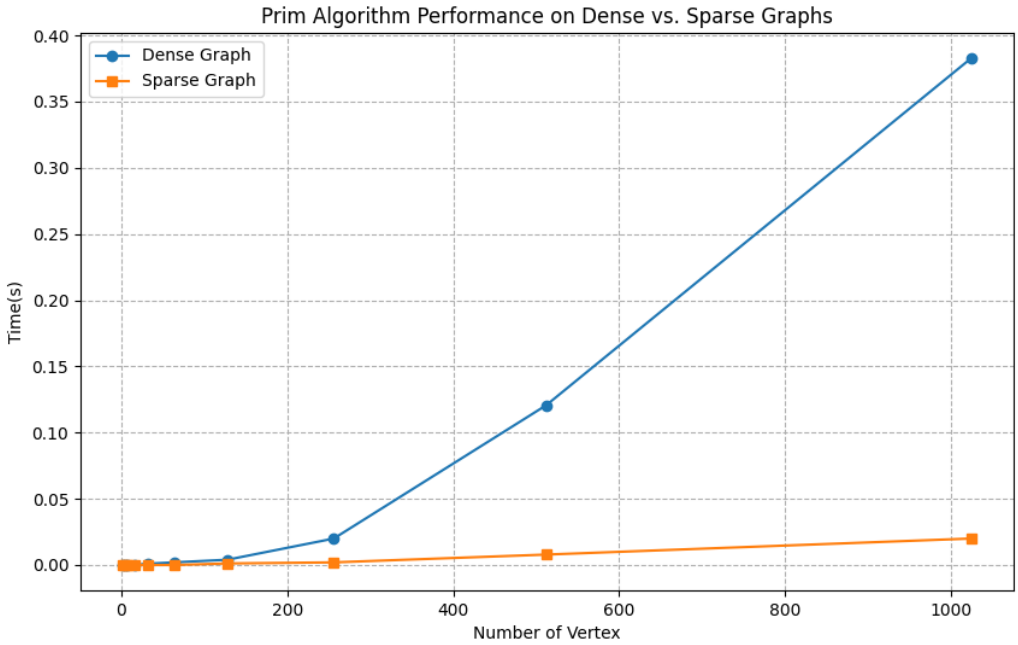
성능테스트

이것도 덴스한것과 스펄스한 걸 비교했다. 2의 거듭제곱 말고 10의 거듭제곱으로 하려고 한다.

음, 컴퓨터가 죽으려고 해서 4의 거듭제곱으로 한다.

음, 이것도 안될 것 같으니 2의 거듭제곱으로 한다. 크루스칼과 비교할 겸…. 해서

그런데 자리가 없으므로 뒷장에 첨부한다.



크루스칼보다 덴스한 그래프에서 괜찮은 성능을 보인다. 그래도 어쩔 수 없이 스펄스한 것보다는 오래 걸린다.

크루스칼 vs 프림

(위 둘 그래프 요약 / 간선이 1024개)

크루스칼: 덴스할 경우 대충 1초 / 스펄스할 경우 0,05초

프림 : 덴스할 경우 대충 4초 / 스펄스할 경우 0.05초 아래

스펄스할 경우의 차이를 보는 것보다, 덴스할 경우를 보면 차이가 더 잘 보인다. 프림 알고리즘의 경우 크루스칼 알고리즘보다 간선의 수 영향을 적게 받는다.

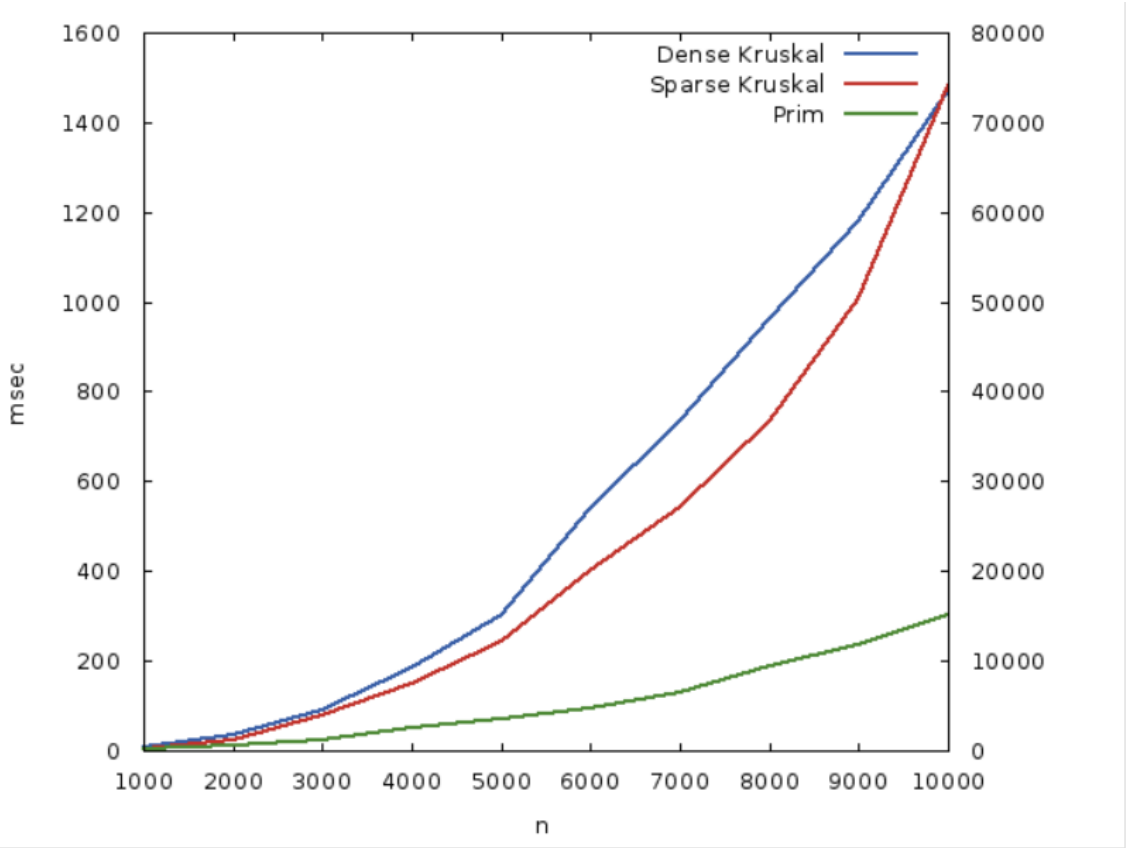
알아본 바에 따르면 크루스칼이 정점의 수가 적을 때 성능이 더 좋다고 하는데, v가 충분히 크지 않아서인지 프림이 더 좋게 나왔다.

스펄스할 경우 두 그래프를 비교하는 그래프를 그리려고 했는데, 차이가 잘 보이지 않았다. 간선의 수를 늘려보았으나 컴퓨터가 강종을 때려버려서 실패했다.

아쉬운대로 해외 글을 하나 찾았다.

https://fedelebron.com/a-dense-version-of-kruskals-algorithm

내가 흥미롭게 본 건 아래 그래프이다



Benchmarks on dense graphs between sparse and dense versions of Kruskal's algorithm, and Prim's algorithm.

가장 명확한 사실은 N이 커지면 덴스한 크루스칼은 덴스한 프림을 절대 이길 수 없다는 것이다. (글을 보면 덴스한 프림이라고 써놨음.) 그리고 생…각보다는 스펄스한 크루스칼이랑 덴스한 크루스칼이랑 큰 차이가 없다는 거?

정점의 개수를 더 늘려서 덴스한거 스펄스한거 각각 직접 비교해 봤으면 더 확실했을텐데, 노트북이 죽으려고 해서 할 수 없었다는 게 아쉬울 따름이다.

* 프림과 크루스칼 차이에 대한 정보는 아래 geeksforgeeks에서 많이 얻었다.

https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/