202301558 김결

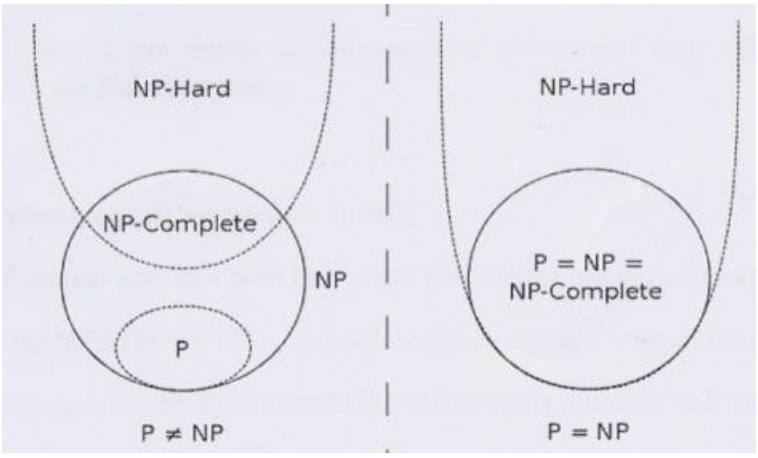
10주차 과제 / TSP

1. 근사 알고리즘?

TSP에 대한 개념이 이해되질 않아서 우선 근사 알고리즘…? 에 대해 정리하려고 한다. 직관적으로 말하자면 '최대한 비슷한 걸 찾아보기' 이다.

10주차 수업에서 배운 NP-완전 문제 같은 경우 현재로써는 알맞은 알고리즘이 개발되지 않았다. 이를 해결하기 위해 guessing, verification(추측과 확인…? 해외 포럼에서 찾은 거라 한국어 용어를 모르겠다.)의 과정을 거쳐야 한다고 한다. 여기서 Guessing이란 해를 하나 추측하는 것이고, Verification은 이 해가 맞는지 검증하는 것이다. 한 글은 이 행위는 다항식 시간 내에서 해결 가능한 다른 문제를 풀어내는 것이라고 표현하고 있었다. 만약 다항식 시간 내에서 해결 가능한 해가 나타나고 검증이 완료된다면 NP-완전 문제는 해결된다. NP-완전 문제의 특징으로, 다항식 시간 내에서 해가 도출된다면 모든 문제를 다항식 시간 내에서 해결할 수 있게 된다고 한다. 왜 그런지는…. 알아보다가 포기했다. 뭐라는지 모르겠다. 세상에.

찾아보다보니 NP-hard(난해)와 NP-complete(완전)이 자주 나오길래 차이를 정리해본다. 아주 간단하가 NP-hard는 다항식 시간 내에서 풀 수 없고 지수 시간 내에서 해결 가능한 문제, NP-complete는 완전한 답을 찾을 수는 없지만 문제를 축소(?) 하여 다항식 시간 내에 근사된 해를 낼 수 있는 문제라고 한다. 즉, NP-complete는 NP-hard이자 NP문제이다.



이걸 이해하는데 도움이 되었던 그림이다. 많은 수학자들이 P=/=NP일 거라고 생각하고 있으므로 왼쪽 그림으로 받아들였다. P=NP임을 증명하면 오른쪽 그림이 될 것이다.

자, 다시 돌아가서 NP-complete 문제에 한하여 우리는 최적해에 가까운 근사해를 찾을 수 있다. '근사해' 라는 단어만 봤을 때는 제일 좋은 해를 찾는건가보다~ 라고 생각했다. 그런데 조금 찾아보니 이 해는 항상 가장 효율적인 방법을 보장하지 않는다고 한다. 왜냐하면 그 방법은 아직 알려지지 않았기 때문이며, 다항식 시간 내에서 가능한한 괜찮은 애를 골라내는 것뿐이기 때문이다.

202301558 김결

근사 알고리즘의 특징을 정리하면 아래와 같다고 한다,

- An approximation algorithm guarantees to run in polynomial time though it does not guarantee the most effective solution.
- An approximation algorithm guarantees to seek out high accuracy and top quality solution(say within 1% of optimum)
- Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time

요약: 최고로 좋다라고 말할 순 없지만 다항식 시간 내에서 풀 수 있다.

2. Traveling Salesman Problem

이번주 주제인 TSP이다. 요약하자면 비용이 가장 적게 드는 해밀턴 경로를 찾는 것이다. 해밀턴 경로는 이산수학시간에 배웠던 개념인데, 다시 출발점으로 돌아오는 경로를 말한다. 이게 왜 다항식 시간 내에 해결이 불가능한지부터 알아봤다. 사람의 입장에서 그다지 많은 계산을 필요로 하지 않는 것 같아서 왜 NP인지부터 이해해보았다.

만약에 N개의 정점이 있고 모든 정점이 연결된 그래프인 완전그래프가 주어졌을 때, 시작 정점을 제외하면 N-1! 번의 계산을 해야 한다.(마지막 경로가 고정되기 때문에 그렇다고 하는데 N! 으로 표현하는 곳도 꽤 많았다.) 다항식으로 표현되려면 N^k의 형태를 보여줘야 하는데, 팩토리얼은 그렇지 않기에 지수시간대로 본다고 한다. 따라서 돌아다니는 판매자 문제…. 가 아니라 한국말로는 외판원 문제는 NP문제가 된다.

또, 해당 그래프는 양방향 그래프여야 하고, 삼각 부등식 특성을 만족해야 한다. 두번째가 뭔 소리냐 하면, 삼각형에서 가장 긴 변의 길이 > 나머지 두 변의 합 인 것을 말하는 건데, 어딘가를 경유해서 가는 것보다 바로 가는게 더 짧아야 한다는 뜻이다.(A->B > A->C->B) 근데 경유해서 가는게 더 짧으면…. 어떻게 되려나…. 더 못푸는 문제가 되려나….

이걸 풀기 위해 제시된 방법은 여러가지가 있는데 분기 한정법, 크로스토피드, 등등 많았다. 그나마 강의 자료에 나와 있는 애가 이해할만 했다.(사실 너무 어렵습니다. 살려주세요.) 방법은 아래와 같다.

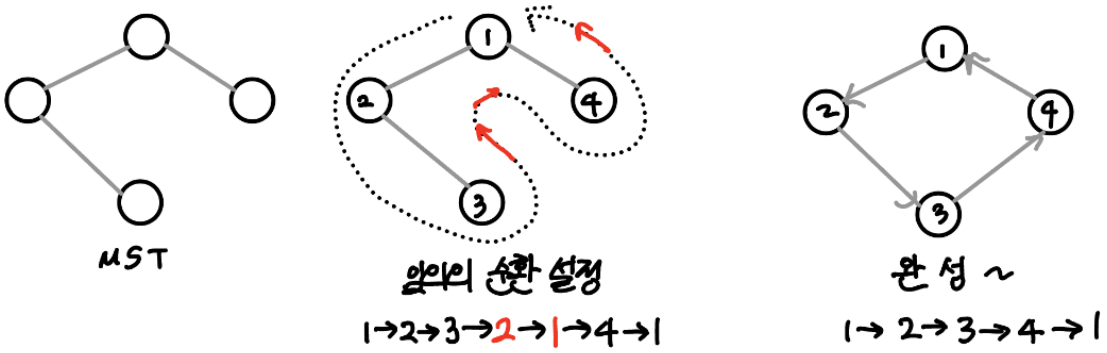
1. 완전그래프 만들기

2. 프림 또는 크루스칼을 이용하여 최소신장트리 만들기

3. 경로상에서 다시 방문하는 정점을 건너뛰기

1, 2는 뭐 알겠는데, 3번이 뭔지…. 왜 해야 하는지…. 머리로 굴려 보다가 안 돼서 손으로 했다.

202301558 김결



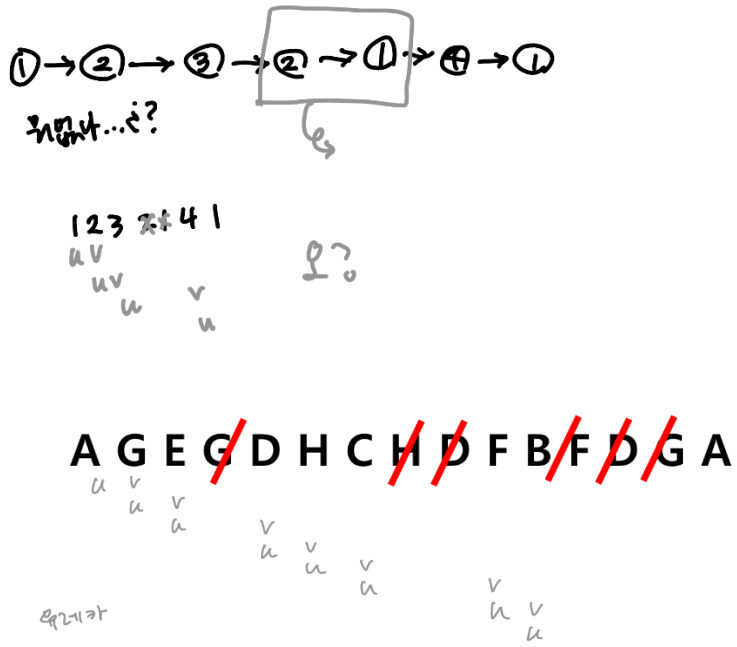
놀랍게도 MST에서 중복방문 노드만 제거했더니 해밀턴 경로가 발견되었다. 신기하다.

아니면 무식하게 n!(또는 n-1!) 번 돌아보는 방법도 있을 것이다. 진짜 최적해를 구할 수 있는 방법이며, 노트북이 아주 좋아하는 방법이다.

3. 구현하기

3-1. tsp

몇시간 고민하다가…. 그냥 고수들이 짠 코드를 보고 칠까 고민도 했다. 그러다가 정말 대박인걸 찾아냈다. 사실 별건 아닌데 웬일로 혼자 떠올려서 놀라웠다.



이게 뭐냐면 한번 방문한 정점은 경로에 새로 추가되지 않는다는 점을 생각하다가 찾아낸 규칙이다. 그러니까 isVisited를 u, v에 대해 각각 만들어놓고 mst에서 뽑아낸 u, v의 방문 여부를 찍는 거다. 예를 들어 1->2가 들어왔을 때 u와 v에 각각 1, 2가 방문했음을 기록하고, 다음에는 2->3이 들어올

202301558 김결

텐데 이때 2번은 u에 아직 기록되지 않았기 때문에 이때 u에 방문 기록을 남길 것이다. 3은 v에 방문 기록을 남긴다. 이후 3->2가 들어오면 u에 3이 기록, v는 이미 2에 기록되었으므로 넘어간다. 여기서 주의해야 할 것은 다시 태어난 곳으로 돌아가야하니까 마지막에 강제로 append 해서 경로의 첫번째 인덱스(시작점)를 추가해줘야 한다는 것이다. 그리고 나는 파이썬의 콘솔 창이 너무…. 뭐랄까 못생겨서 matplotlib으로 그래프를 그리려고 한다. 제발 좋은 아이디어이기를 바란다.

```
# 정점이랑 인접행렬을 만들어
# 크루스칼 복붙해서 mst 를 받아

# def 해서 티에스피(mst)
# 경로를 저장할 리스트
# 방문 여부를 알려줄 u, v 의 visited 를 F 로 세팅
# mst 의 모든 간선에 대해서 for
# u, v 를 뽑아. 간선 edge 는 튜플
# u 가 방문 이전이면 u 에 추가, 방문기록
# v 가 방문 이전이면 v 에 추가, 방문기록
# 태어난 곳으로 돌아가기 경로[0] 추가
# 경로 리스트를 반환
```

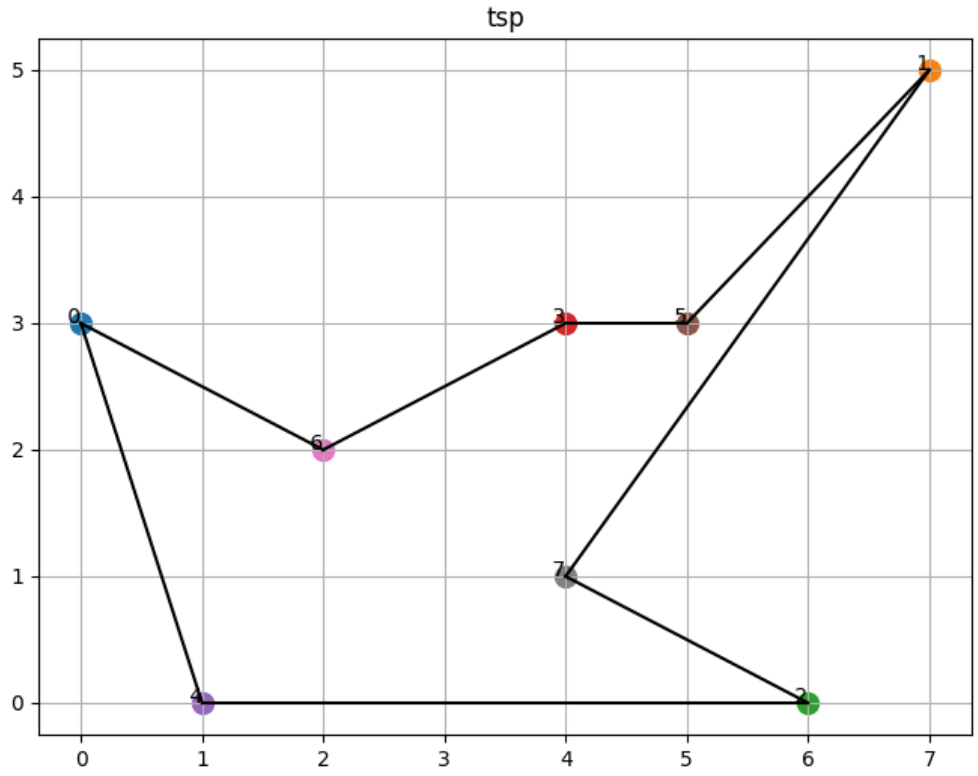
수도코드대로 짰는데, 시작 지점 설정에 문제가 있어서 챗지피티가 다시 짜줬다…. 혼자 100퍼센트 짜는건 언제 해보나….

```
def tsp(v):
    # 크루스칼 복붙해서 mst 를 받아
    mst = kruskal(v)
    # 경로를 저장할 리스트
    path = []
    # 방문 여부를 알려줄 u, v 의 visited 를 F 로 세팅... 했더니 이상한거 떠서...
    # 챗지피티가 set 쓰라고 알려줌... 왠지는 모르겠음...
    visited_u = set()
    visited_v = set()

    # 챗지피티가 만들어준... 코드... 히히...ㅜㅜㅜㅜㅜㅜ
    def visit(node, from_node):
        # Depending on direction, mark different sets
        if from_node in visited_u:
            visited_v.add(node)
        else:
            visited_u.add(node)
        path.append(node)
        for u, v in mst:
            if u == node and v not in visited_v and v not in visited_u:
                visit(v, u)
            elif v == node and u not in visited_u and u not in visited_v:
                visit(u, v)

    # Start visiting from the first node
```

```
    visit(0, None)
    path.append(path[0])
    return path
```

결국 혼자 한건 위에 네줄이다…. ㅜㅜㅜ 너무 어렵다….

아무튼 결과는 아래처럼 나왔는데 강의자료랑 달라서 틀린 건 확실하다. 챗지피티가 tsp는 괜찮다고 하는데 뭘까…. 진짜 뭘까….




```
Path:  [0, 6, 3, 5, 1, 7, 2, 4, 0]
```

점에 번호 붙이는 법은 챗지피티가 알려줬다.

```
for i, point in enumerate(v):
    plt.scatter(*point, s=100)
    plt.text(point[0], point[1], f' {i}', ha='right')
```

이렇게 한다고 하는데…. 그렇다고 한다…. 나중에 써볼 일이 있겠지.

아, 다 짜놓고 총가중치 표시가 안 됐다는 걸 깨달아서 sum을 사용해 반복문을 돌면서 가중치를 더해주는 함수는 따로 짰다.

```
total = 0
for i in range(len(path) - 1): # len(path)까지 안 돌려도 된다고 한다.
    w = dist(v[path[i]], v[path[i+1]])
    total += w
```

202301558 김결

이렇게

`23.698908717413936` 답이 맞는지는 모르겠다. 틀렸겠지 뭐…. ㅜㅜ….

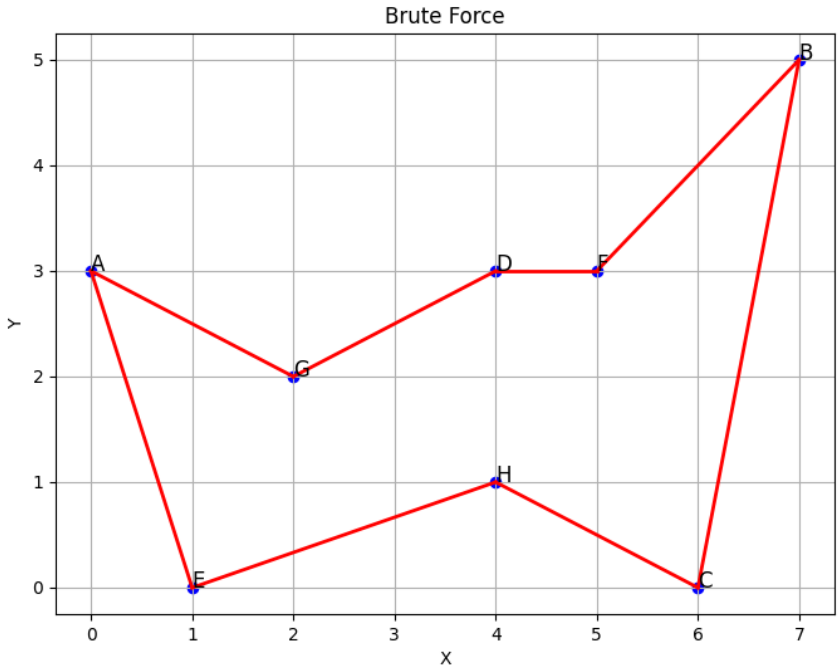
시간복잡도: 강의자료에 나온 대로 크루스칼(or 프림)과 같다. 중복을 제거하는건 O(n)이라 영향을 주지 못하는 듯

3-2. 최적해 구하기(브루트포스 이용)

이제 브루트포스로 n!번 돌면서 최적해를 구할 것이다. 이걸 하기 위해서는 순열을 쉽게 생성해주는 itertools 를 사용하는게 좋다고 한다. 오히려 이게 정보 찾기가 엄청 힘들어서 영어로 서치해도 잘 안 나오길래 그냥 혼자 하면서 무한 챗지피티를 돌렸다.

```
import itertools

def total_d(path, d): # 총 가중치
    t = 0
    for i in range(len(path)-1):
        t += dist(d[i], d[i+1]) # 거리
    t += dist[d[-1], d[0]] # 시작 정점도 넣어야지...
    return t

def tsp_bruteforce(v, name):
    n = len(v)
    distances = np.zeros((n, n)) # n by n의 행렬


    for i in range(n):
        for j in range(i, n):
            dist[i, j] = distances[j, i] = dist(v[i], v[j])

    min_d = float('inf') # 최소 경로를 발견할때마다 여기 업데이트를 해줄 거야
    min_path = None

    for path in itertools.permutations(range(n)): # 이게 순열 그거. 모든 순열을
만들어줌
        d = total_d(path, distances) # 총가중치 업데이트
        if d < min_d: # 최소경로를 찾았으면
            min_d = d # 최소가중치 업데이트
            min_path = path # 최소경로를 기록하려면 이렇게 하래


    return min_path
```

202301558 김결



<- 결과

조금 더 서치해보니까 브루트포스 방법으로 tsp 해놓은 코드를 발견해서 조금 더 만졌다. ㄴ그런 김에 정점 이름도 넣어봤다.

브루트포스 결과로 나온 총 가중치

```
21.9602058911751
```

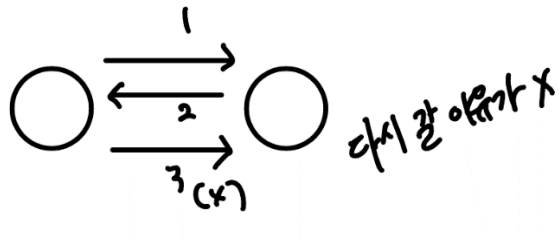
이게 최적해이다. 근사해보다 확실히 저렴하다? 더 좋다.

4. 근사 비율 계산하기

최소신장트리의 총 가중치를 M으로 둘 때, 최적해의 값은 M보다 크다: 이게 무슨 말인지 내가 이해한 대로 말하자면, tsp의 해에는 무조건 태어난 곳으로 돌아가는 간선이 추가되게 되고, 방문한 정점을 또 방문해야 할 수도 있기 때문에 M보다 클 수 밖에 없다. 정도로 이해했다. 맞는지는 모르겠다.

그리고 근사해의 값은 2M보다 작다: 이것도 내가 이해한 대로라면, 모든 간선을 두번 이상 지나는 건 불가능하다.

202301558 김결



왜인지 말로 설명하는게 어려워서 그렸다.

그리고 근사 비율을 계산할건데, 근사해 / 최적해 를 하면 된다. 별거 없다.

근사해: 23.698908717413936

최적해: 21.9602058911751

근사 비율: 1.07917516….

너무 어렵다…. 진짜 너무 어렵다…. 살려주세요….

202301558 김결

9주차 과제 / 동적 계획 알고리즘 – 모든 쌍 최단 경로 문제

였던 것... 삭제하긴... 아까워서...ㅜㅜ

모든 쌍 최단 경로 문제는 말 그대로 모든 쌍에 대해서 최단 경로를 구하는 문제이다. n개의 점에 대하여 n x n의 표를 채우는 거라고 생각하면 된다. 물론 대각선을 기준으로 값이 똑같기 때문에 위쪽에만 채우면 된다.(강의 자료 7쪽 그림처럼)

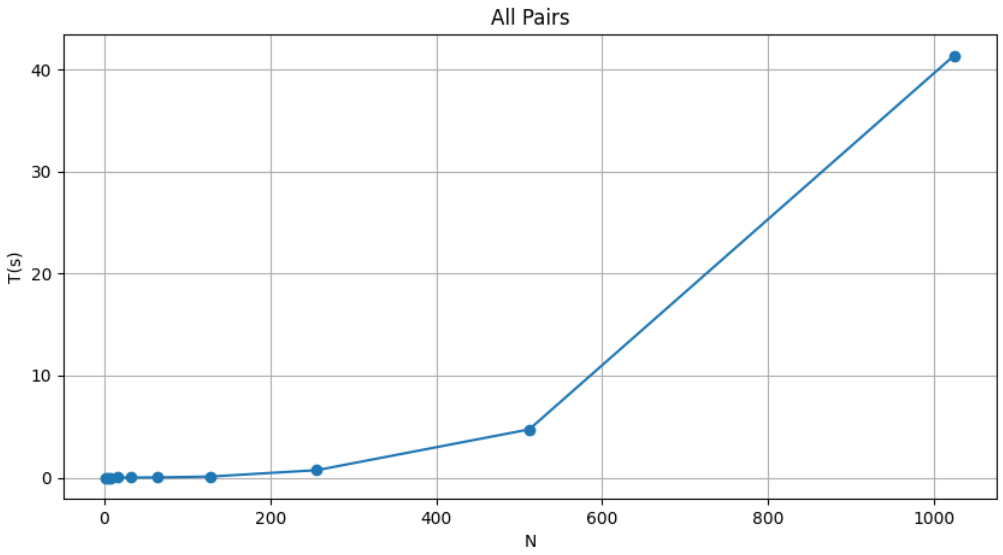
1. 다익스트라 알고리즘을 n번 사용하기

앞서 배운 다익스트라 알고리즘은 한 점에 대하여 최단경로들을 찾아주는 알고리즘이다. 표로 생각하면 한 열을 채우는 것이다. 그러면 이걸 n번 돌려주면 표가 다 채워질 것이다. 당연하게도 O(n^2)짜리 알고리즘을 n번 반복하니까 시간복잡도는 O(n^3)이다.

강의자료상에서 바로 다른 알고리즘이 나오길래 구현은 건너뛰려고 했는데, O(n^3)짜리 알고리즘의 성능 분석 그래프를 그리면 얼마나 노트북이 좋아할지 궁금해서 해 보았다.

```
# 그래프를 받아오는 def
# 배열에 점들을 집어넣기
# 거리를 저장해줄 배열
# 모든 점들에 대해서 for, 다익스트라 호출
# 리턴
```

의사코드인데 너무 간단해서 바로 성능 그래프를 첨부한다.(다익스트라는 깃허브에서 주웠다)

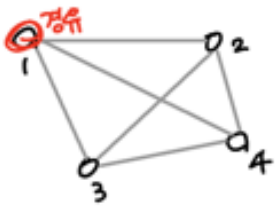


2의 13승까지 돌려보려고 했는데 역시 강종을 때려버려서 실패했다. 1024까지가 한계인 것 같다.

202301558 김결

2. 플로이드-워샬 알고리즘

강의 자료로는 이해가 잘 안 돼서 깃허브에서 도움을 받았다. 이해한 대로 필기를 해 보았다.



한 경유점을 골라

1번을 거쳐가는 모든 경로를 바로 가는 경로와 비교(min 쓰면 되겠다.)

그러니까 2→1→3, 2→1→4를 2→3, 2→4와 비교하라는 뜻.

그리고 경유점으로 선택된 애는 시작지점, 끝지점이 될 수 없음 (1→1→2, 2→1→1 안된다고)

시작지점이 아니라 경유점을 선택한다는게 포인트인듯.

	1	2	3	4	
1	0	vv	vv	vv	1 경유
2	vv	0	vv	vv	2 경유
3	vv	vv	0	vv	3 경유
4	vv	vv	vv	0	4 경유

그리고 경로를 만들 수 없는 경우 무한대로 설정한다고 한다

아래는 내가 적은 의사 코드이다

```
# 가중치 정보, 노드 정보를 담은 배열을 받는 def
# n개의 경유점에 대하여(아우터 루프)
# n개의 시작점에 대하여(이너 루프)
# n개의 도착점에서(이너이너 루프...?ㅋㅋ)
# 바로 가는 길 vs 경유점 거쳐 가는 길
```

그리고 강의 자료를 확인했는데 아주 비슷해서 바로 구현했다. 강의 자료에 나온 대로 간단해서 편하게 구현할 수 있었다.

```
inf = float('inf')
```

202301558 김결

```
n = 5
D = [[inf] * n for _ in range(n)]

for i in range(n):
    D[i][i] = 0

D[0][1] = 4
D[0][2] = 2
D[0][3] = 5

D[1][2] = 1
D[1][4] = 5

D[2][0] = 1
D[2][1] = 3
D[2][3] = 1
D[2][4] = 2

D[3][0] = -2
D[3][4] = 2

D[4][1] = -3
D[4][2] = 3
D[4][3] = 1

def allpair(D):
    for k in range(0, n):
        for a in range(0, n):
            for b in range(0, n):
                D[a][b] = min(D[a][b], D[a][k] + D[k][b])
    return D

print(D)
allpair(D)
print(D)
```

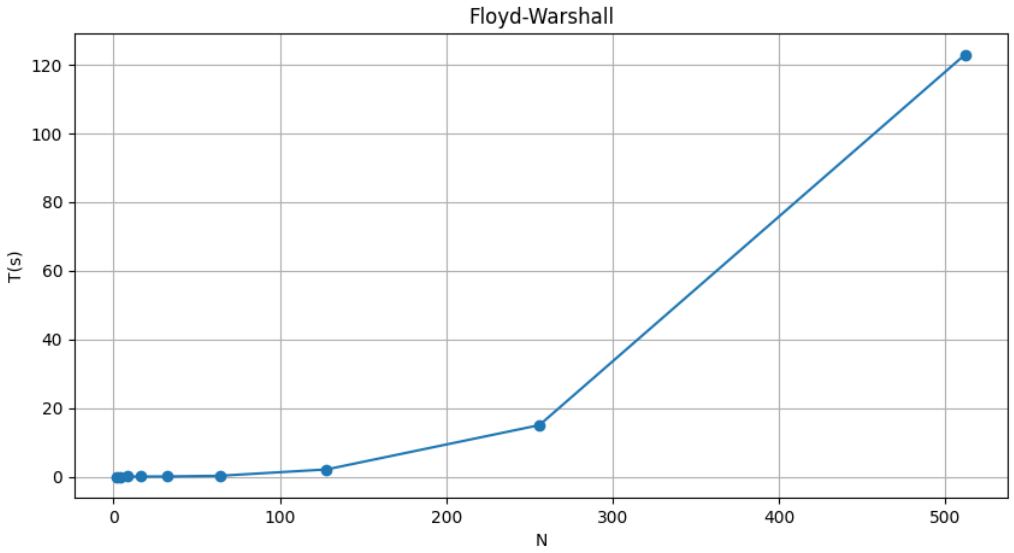
(강의자료와 똑같이 하려고 했는데 강의 자료엔 0번 노드가 없다는걸 지금 깨달았다.)

출력 결과도 강의자료와 똑같이 예쁘게 잘 나왔다.

```
[[0, 4, 2, 5, inf], [inf, 0, 1, inf, 5], [1, 3, 0, 1, 2], [-2, inf, inf, 0, 2], [inf, -3, 3, 1, 0]]
[[0, 1, 2, 3, 4], [0, 0, 1, 2, 3], [-1, -1, 0, 1, 2], [-2, -1, 0, 0, 2], [-3, -3, -2, -1, 0]]
```

이제 성능을 분석해볼 건데, 코드를 보면 누가 봐도 O(n^3)짜리이다. For이 세 개 있으니까 당연한 거다. 뭐 알아볼 것도 없다.

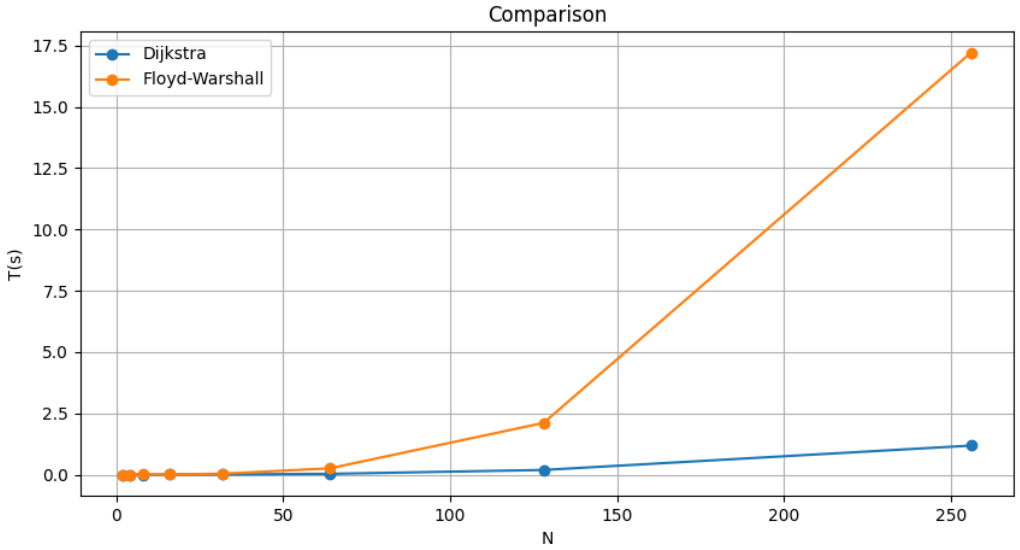
202301558 김결



얘도 1024개까지 돌려보려고 했는데 하루 종일 무한 작동중이라서 512개까지로 줄였다. n^3 짜리는 정말 엄청나다는 교훈을 얻었다... 세상에...

3. 다익스트라 * n vs 플로이드 워셜

이건 그냥 혼자 해본 공부인데, 둘 다 O(n^3)짜리라면 그래프가 완전히 겹치거나 비슷하게 나오려나? 싶어서 같이 그려 보았다. 그런데 아래처럼 꽤 속도 차이가 나길래 원인을 좀 찾아보았다.



(우선 내 생각에는 N수가 작은 게 가장 큰 원인 같기는 하다.)

https://www.researchgate.net/post/Floyd-Warshall_algorithm_or_Dijkstras_Algorithm

202301558 김결

여기에 달린 답변들을 읽어보고 서치해본 결과, 플로이드 워셜 알고리즘 같은 경우에 희소그래프에서 보여주는 성능이 좋지 않다고 한다. 스펄스할 경우 힙을 사용하는 다익스트라와 달리 플로이드 같은 경우엔 for을 세번 돌면서 무의미한 계산을 해야 하기 때문인 것 같다. 플로이드 워셜 그래프의 성능을 구할때 랜덤 그래프 생성 함수를 주워서 사용했는데, 이때 스펄스한 그래프들이 만들어졌을 수도 있겠다. 실제로 몇 번 더 돌려 봤더니 간격이 줄기도 하고 더 심해지기도 하고 그랬다. 또, 메모리 공간을 다익스트라보다 많이 잡아먹는다는 단점도 있었다.

(https://www.geeksforgeeks.org/time-and-space-complexity-of-floyd-warshall-algorithm/)

반대로 플로이드 워셜은 훨씬 넓은 범위에서 사용할 수 있고 적용하기도 편하다고 하는데, 가장 큰 강점은 음수를 가질 때도 적용이 가능하다는 거다.

https://www.baeldung.com/cs/dijkstra-negative-weights

이 글은 다익스트라 알고리즘은 음의 가중치가 있을 때 무한 싸이클의 생성과 이상한 결과를 도출할 수 있음을 안내해 주고 있다. 반면 플로이드 워셜 같은 경우 힙을 사용하는게 아니라 모든 경우에 대해 단순히 비교연산을 하기 때문에, 방문한 정점을 다시 방문해서 최솟값을 찾아내는 다익스트라와는 다르다.

(https://stackoverflow.com/questions/22891420/why-do-all-pair-shortest-path-algorithms-work-with-negative-weights 이 글을 참고했다)

마지막으로 공부하다가 찾은 페이퍼(https://arxiv.org/pdf/2109.01872)에서, 플로이드 워셜 알고리즘을 발전시킬 수 있다는 정보를 찾았다. 무한대로 설정되어 있는(경로를 만들 수 없는) 경우에 대해 비교하는 연산을 없애기, 그리고 더 발전시켜서 이너루프들이 도는 수를 조절하는 방법으로 성능 향상을 볼 수 있다고 한다. 무슨 소리인지 완벽하게 알아들을 수는 없었지만... 이런게 있다는 걸 알아간다.