

Binary Heap Adopted Dijkstra

Runtime Analysis /

2026/04/26

Seungjun Lee

Dijkstra : 최단 경로 탐색 알고리즘.

원리 = 시작점에서 시작하여 현재 위치에서 갈 수 있는 인접 노드들에 대해 재산을 거쳐가는 경로가 기존에 알려진 경로보다 빠르다면 업데이트.

이후 방문하지 않은 노드 중 경로가 가장 빠른 노드를 선택하여 위의 과정 계속 반복.

더 이상 유한한 시간에 갈 수 있는 노드가 없을 때까지 반복.

시간복잡도

- Pure Dijkstra

노드의 수 = V , 간선의 수 = E , 그래프 밀도 = $D = \frac{E}{V \times (V-1)}$

$\Gamma_{cur} = 0$

While True :

← V th iteration

cur → visited

for neighbor of cur :

← E_i th iteration
cur = V_i 인 때,

if cur → neighbor < neighbor :

update neighbor

└ cur = unvisited min node

간단한 pseudo-code

Time complexity

$$= O(V(E_i + V))$$

$$= O\left(\sum_i E_i + V^2\right)$$

$$= O(E + V^2)$$

min 찾는 데에
 $O(V)$ 소요

이와같이 Pure-Dijkstra의 시간복잡도는 $O(V^2 + E)$ 이다.

이렇게 되면 V 번의 iteration마다 $\min$ 노드를 찾는 데에

$O(V)$ 가 소요된다. 이를 개선하기 위해 Priority Queue (우선순위 큐)를 Dijkstra에 적용하였다.

Priority Queue는 Heap structure로 구성되어 있다.

Heap이란 최소/최대값을 빠르게 추출할 수 있는 데이터 structure이다.

이 structure가 지원하는 operation들은 이와 같다.

— Add : 새로운 노드를 Heap에 추가하는 동작

— Extract-min : Heap에서 가장 작은 값을 가진 노드를 추출하고
Heap에서 자르는 동작

— Decrease-key : 기존에 존재하는 노드의 값을 줄이는 동작

* 시간 최소값을 다루는 Heap의 경우이고, 최대값을 다루는 경우에는

반대로 Extract-max와 Increase-key가 있다.

Heap도 여러 종류가 있는데 대표적이고 가장 많이 사용되는 것은

Binary Heap이다. 이론상 더 빠른 Heap도 있지만 실제 runtime 상

Binary가 가장 빠르고, 구현도 간단하기 때문이다.

Binary Heap 의 operation 별 시간 복잡도는 이다 같다.

Operation	Time complexity
Add	$O(\log V)$
Extract-min	$O(\log V)$
Decrease-key	$O(\log V)$

이 이유는 Binary Heap은 완전 이진 트리 구조를 기반으로 한다.

또한 자식노드의 값은 부모 노드의 값 이상이어야 한다. (최소 힙 기준)

Add를 하는 경우 새 노드를 제일 마지막에 추가하고,

Extract-min의 경우 최상단 노드를 빼 리렌하고 제일 마지막 노드를 빈 최상단 자리에 넣으며,

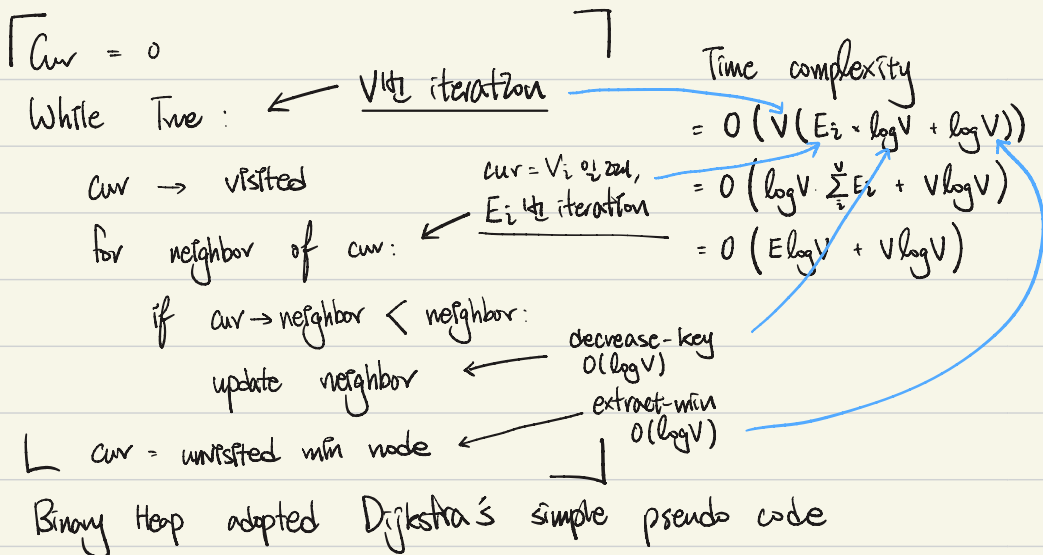
Decrease-key의 경우 해당 노드의 값을 변경한 후,

모두 변경 지점을 기준으로 부모와 자식간의 크기 관계가 맞을때까지 변경하는 과정을

수행한다. 이때 V 개의 노드가 있을때 이진노드의 높이는 $\log_2 V$ 이기 때문에,

시간 복잡도가 모두 $O(\log V)$ 가 되는 것이다.

Binary Heap을 Dijkstra에 적용하면 시간복잡도가 달라진다.



따라서 시간복잡도는 $O(V \log V + E \log V)$ 이다.

이러므로 Runtime = $V \log V \cdot k_1 + E \log V \cdot k_2$ 로 표현할 수 있다고 생각할 수 있지만 사실 그렇지 않다. add에 대해서는 변하지 않으며, decrease-key도 V 번 발생한다고 가정하기 때문이다.

실제로는 Dijkstra는 Add + Extract-min + Relax-attempts + Decrease-key로 구성된다.

- Add는 초기 queue에 노드들을 넣는 것을 뜻한다.

총 V 개의 node 넣기 때문에 $V \cdot \log V \cdot k_1$ 으로 Add 총시간 고려 가능.

하지만, 실제로는 처음에 넣는 모든 노드의 값이 ∞ 이기 때문에 사실 $O(1)$ 로 처리됨.

- Extract-min도 총 V 번 실행되기에 $V \cdot \log V \cdot k_2$ 로 나타낼 수 있다.

사실 Extract-min을 계속 진행해보면 노드 수가 1씩 줄기에 걸리는 시간으로

$$\log(V) \cdot k_2 + \log(V-1) \cdot k_2 + \log(V-2) \cdot k_2 + \dots \log(1) \cdot k_2$$

$$= k_2 \cdot \log(V(V-1)(V-2) \dots 1) = k_2 \cdot \log(V!) \text{ 가 된다. 그러나 여기에서는 } V \cdot \log V \cdot k_2 \text{로 계산함}$$

- Relax-attempts 는 E 번 실행된다.

따로 Heap 관련 operation은 없어 총 소요시간은 $E \cdot k_3$ 로 나타낼 수 있다.

- Decrease-key 는 relax.success (RS) 만큼 실행된다.

따라서 총 시간은 $rs \cdot \log V \cdot k_4$ 가 된다.

정리하면, Runtime = Add + Extract-min + Relax-success + Decrease-key

$$= V \log V \cdot k_1 + V \log V \cdot k_2 + E \cdot k_3 + RS \cdot \log V \cdot k_4$$
$$= V \log V (k_1 + k_2) + E \cdot k_3 + RS \cdot \log V \cdot k_4$$

이때 V, E, RS 를 넣고 회귀방식으로 k_1, k_2, k_3, k_4 를 구할 수 있는 것이라 생각할 수 있다.

다만, 아래의 문제점들이 있다.

— k_1 과 k_2 를 식에서 구할 수 없어 해를 묶어 생각해야 한다.

그러나 귀의 목표는 전체 runtime 예측이기 때문에 k_1 과 k_2 를 개별적으로 구하지 못해도 상관없다.

— 더 중요한 문제는 다중공성이다. (Add + Extract-min)과 Decrease-key에 있는 $\log V$ 로 인해 두 항의 구분이 어려워져 회귀방식이 잘 되지 않을 수 있다.

이를 해결하기 위해 V 가 동일한 곱셈의 결과만을 가지고 병합한 이후, V 번 결과의 양상을 분석해야 한다.

가운 Runtime = $\overset{UC_1}{V \times \log V \cdot (k_1 + k_2)} + \overset{UC_2}{E \times k_3} + \overset{UC_3}{RS \times \log V \cdot k_4}$ 이다.

● : operation
실행 횟수

● : operation
unit cost

특정한 V 에 대한 runtime 분석 시 중요한 점은 **하루씩** operation unit cost
부분에 있는 V 들이 상수가 된다는 것이다.

$$\text{let } UC_1 = \log V \cdot (k_1 + k_2)$$

$$\text{let } UC_2 = k_3$$

$$\text{let } UC_3 = \log V \cdot k_4$$

$$\text{Runtime} = V \cdot UC_1 + E \cdot UC_2 + RS \cdot UC_3 \text{ 이 된다.}$$

각 노드별로 V, E, RS , Runtime 을 넣고 회귀분석을 하면 UC_1, UC_2, UC_3 을 구할 수 있다.

사실상 UC_2 는 상수이고, UC_1 과 UC_3 은 $\log$ 증가를 보여야 한다.

이제 UC_1 과 UC_3 은 V 에 따라 $\log$ 증가해야하기에 다시 한번 UC_1, UC_3 을 각각 V 에 대해

크고 회귀를 적용하면 모든 상황에서의 각 operation 의 unit cost 를 계산할 수 있다.

- 마지막 문제는 V 와 E 는 Dijkstra를 실행하지 않더라도 빠르게 구할 수 있지만,
 RS 는 Dijkstra를 실행하지 않으면 몇번 호출되는지 알 수 없다. 그렇다면 $RS = V$ 로
계산하기에는 그래프마다 너무 달라져서 심각한 오류 도차 발생할 것이다.
따라서 Dijkstra 실행 없이 원본 그래프 데이터에서 쉽게 측정할 수 있는
값을 만드느 Decrease-key 호출 횟수 RS 를 예측해야 한다.

$$\text{Relax-success-ratio} = \text{Relax-success} / \text{Relax-attempts} \text{ 라고 한다면,}$$

$$\text{Relax-success} = \text{Relax-attempts} \times \text{Relax-success-ratio} \text{ 이다.}$$

$$\text{Relax-attempts} = E \text{ 이고, Relax-success-ratio를 구하면 된다.}$$

Relax-success-ratio (RSR) 은 그래프의 원리나 edge들의 분산에 영향을 받는다

것이라고 생각해서 Δ 를 판단하고, 가설이 맞다면 Δ 번씩으로 RSR을 모킹하여
 RS 를 계산하고, 최종 Runtime을 계산할 수 있다.

이제 이러한 분포를 생성한 데이터가 필요하다. 특히 회귀분석을 하기 위해서는 다양한 조건의 데이터들이 필요한데, 실제 데이터에서 내가 원하는 것들을 뽑아내는 것은 불가능하다. 예를들어 평균 = 100, 분산 = 80 인 Road graph가 필요한데 이런 데이터를 찾는 것은 전혀 불가능하다. 따라서 실제 Road graph를 분석하여 데이터의 분포, 평균, 분산, 밀도 등을 추출한 후 이를 기원으로 변수들을 변형해가며 가상의 그래프를 만들어야 한다.

이를 위해 DIMACS의 USA Road graph data를 분석했다. 분석 결과 edge 값들의 분포는 log-normal에 가까웠으며, 평균은 3000, 표준편차는 4000정도에 밀도는 1×10^{-7} 정도였다.

이를 바탕으로 아래의 조건으로 다양한 데이터를 만들었다. (모두 log-normal 기반)

Variable	Range / Value
V	2000, 4000, 6000, ..., 18000, 20000
D	$1e^{-7}$, $3e^{-7}$, $1e^{-6}$, $3e^{-6}$, ..., 0.01, 0.03
std	1000, 1500, 2000, 3000, 4000, 6000, 8000, 12000, 16000
mean	3000
trials	400

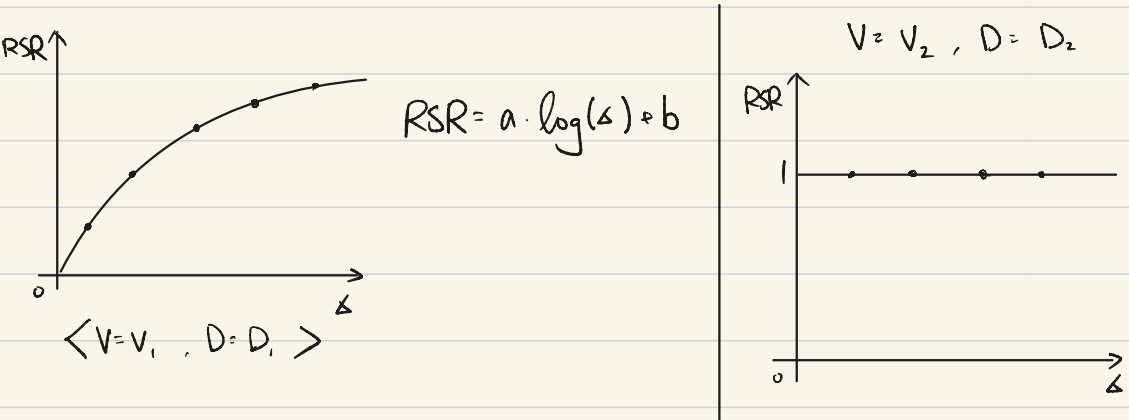
이후 생성한 864000개의 graph에 Binary Heap adopted Dijkstra 생성.

이 과정에서 각 시퀀스에서 Decrease-key가 몇번 시행되었는지를 총 Runtime을 기록하였다.

이제 분석해야할 것은 아래 2가지이다.

- Decrease-key call number
- Operation unit cost

우선 분산과 밀도가 실제로 Decrease-key에 영향을 미치는지 확인하기 위해, RS 분산을 먼저 살펴본다. 이때 $RS = E \cdot RSR$ 이고, E는 양기에 RSR에 영향을 준다. RSR의 분산과 밀도와의 관계를 파악해야한다. 그래서 Δ 와 RSR plot을 V, D 고정된 상황에서 하고, 반대로 D와 RSR plot 하고 V와 Δ 고정했다. 그 결과 Δ 와 RSR은 단변량 2차 관계를 보였다.

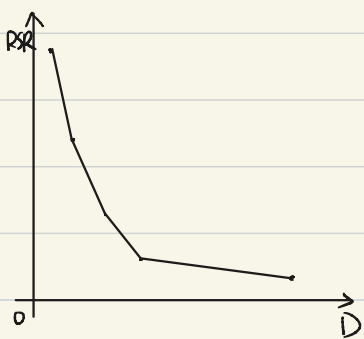


그러나 특정 조건에서는 Δ^2 에 상관없이 RSR이 인 경우로 있었다. 이 원인을 많이 위해 조사해보니 $(V-1) \cdot D$ 즉 Avg-Degree < 1

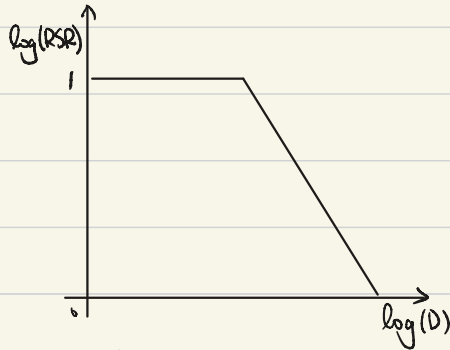
신 상황에서 반사한다는 것을 알게되었다. $(V-1) \cdot D = \frac{E}{V}$ 로

하나의 노드가 평균 몇개의 edge와 연결되어있는지를 뜻한다.

이는 D와 관련이 있는 사항이며, Δ 와 RSR은 로그관계라고 반사하고
D로 넘어갔다.

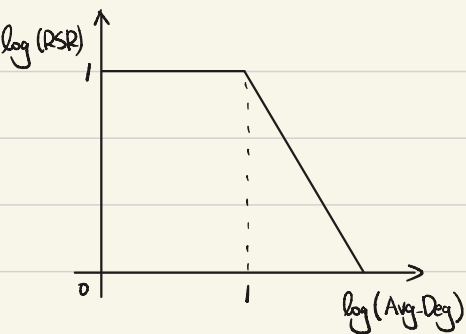


$\langle V = V_3, \Delta = \Delta_3 \rangle$



$\langle V = V_3, \Delta = \Delta_3 \rangle$

D와 RSR의 plot시 양상은 이와 같다. 급격하게 감소하러 완만해지는 것이
power-law 같아 log-log로 변형 후 plot 했다. 그 결과 오른쪽 그래프같이
초기에는 수평이다 어느 시점부터 선형적으로 감소했다. 따라서 전환점 이루는
power-law 관계라는 것이 밝혀졌다. 또한 이 전환점은 역시 $\text{Avg-Deg} = 1$ 이 되는
지점이었다. $\text{Avg-deg} = (V-1) \cdot D$ 인데, V를 고정시켜 봤을 때 (V-1)은
상수이다. 따라서 고정된 V와 Δ^2 에서 Avg-Deg와 RSR을 log-log로
plot 했다.

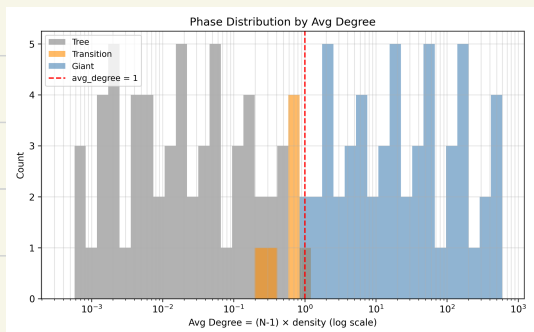


이런 결과가 나왔다. 따라서 $\text{Avg-Deg} \geq 1$ 인 경우에는 Avg-Deg 가 RSR 과 Power-law를 따른다는 것을 알 수 있다.

$$\begin{aligned}\log(\text{RSR}) &= a \times \log(\text{Avg-Deg}) + b \\ &= \log(\text{Avg-Deg}^a \times c)\end{aligned}$$

$$\text{RSR} = c \cdot \text{Avg-Deg}^a \quad (\text{Avg-Deg} \geq 1)$$

이렇게 조절한 두 식을 합치기 $\text{Avg-Deg} \geq 1$ 근처의 전이점에 대해 더 조사해보았다. 그 결과 $\text{RSR} = 1$ 도 아니고 D도 log 관계로 아닌 무규칙적인 구조로 $\text{Avg-Deg} = 1$ 근처에 존재했다. 이를 비롯한 결과, 아마도 같은 결과가 나왔다. 이는 phase-transition 개념과 완전히 동일한 구조였다.



$\text{Avg-Deg} \ll 1$ Subcritical regime

$\text{Avg-Deg} \approx 1$ Critical regime

$\text{Avg-Deg} \gg 1$ Supercritical regime

다시 줄어든 증감한 수를 합쳐 RSR을 나타낸다.

이제 Avg-Deg 다 Δ 는 독립변수이기에 두 식을 곱한다.

$$\begin{cases} RSR = c_1 \cdot \log(\Delta) + c_2 \\ RSR = c_3 \cdot (Avg-Deg)^{c_4} \end{cases} \quad (C \text{ 는 상수})$$

$$\begin{aligned} \Rightarrow RSR &= \{c_1 \cdot \log(\Delta) + c_2\} \times \{c_3 \cdot (Avg-Deg)^{c_4}\} \\ &= Avg-Deg^{c_4} \times \{c_1 c_3 \cdot \log(\Delta) + c_2 c_3\} \\ &= Avg-Deg^a \times \{b \cdot \log(\Delta) + c\} \quad (Avg-Deg \geq 1) \end{aligned}$$

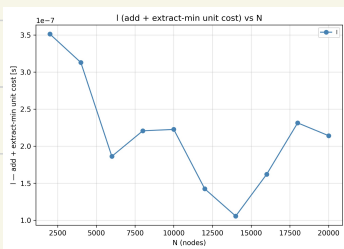
증감한 식으로 Avg-Deg ≥ 1 인 데이터에 non-linear 회귀분석 진행했다.

$R^2 = 0.9947$ 로 정확한 모델링 성공함.

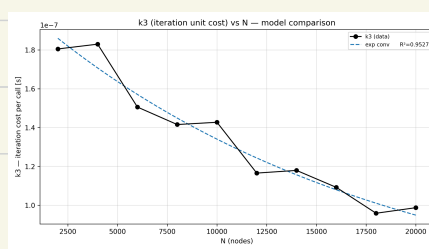
이제 operation unit cost 분석함. 앞서말했듯, 다중공선성 방지 위해 V별로 따로 선형회귀하였다.

$$Runtime = V \cdot VC_1 + E \cdot VC_2 + RS \cdot VC_3$$

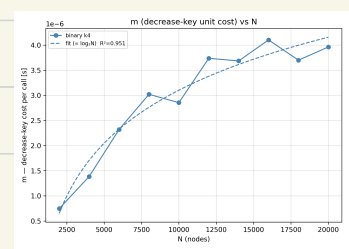
이를 통해 V별 VC 값을 도출함.



VC_1

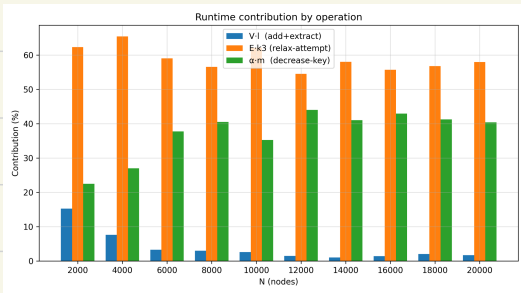


VC_2



VC_3

그 결과 UC_3 는 이론대로 $\log$ 증가하였으나, UC_1 과 UC_2 는 이론을 벗어남.
 우선 UC_1 은 데이터의 양에도 파이프라인이 불가능했다. UC_2 의 경우는 상수여야하나
 지수포화 형태로 감소하였다. 이는 아라 파이프라인의 캐싱 효과로 판권되어 우선
 지수포화 함수로 모델링하였다.



모 각 operation에 쓰는 시간이 전체 Dijkstra runtime에 어느정도
 비중을 차지하는지 보았더니 $(Add + Extract - min) = V \cdot UC_1$ 이 가장 낮았다,
 이다 줄이 큰 영향을 미치지 않고 밀관적으로 커지거나 낮아지는 경향이
 파이프라인 많아 각 V 별 UC_1 값의 평균을 대폭감으로 사용했다.

이름 통해 모든 식과 계수들을 알아낸 후 예측을 했다. 예측결과 데이터가
 커질수록 Runtime의 오류가 컸다. 그러나 RS 예측은 실제 데이터에 대해서도
 상대 오차 0.1 이하를 기록하였다.

$$P(G) = p^m \cdot (1-p)^{nG_2 - m}$$

$$G(n, p) \longrightarrow \text{노드 : } n \text{개}$$

두 노드 간 연결될 확률 : p

↖ 비대칭 그래프의

$$\text{총 노드쌍 개수} = {}_n P_2 = n(n-1)$$

$$\text{총 간선의 개수} = n(n-1) \cdot p$$

이항분포:

$$X \sim B(n(n-1), p)$$

$$P(X=m) = p^m \cdot (1-p)^{n(n-1)-m}$$

$$\begin{aligned} \text{평균 차수} &= \frac{n(n-1) \cdot p}{n} \\ &= (n-1) \cdot p \end{aligned}$$

$$\binom{\binom{n}{2}}{N} = C_{n, N}$$

