Algorithm Term Project (Spring 2017)

Name: 정기빈 Student ID: 2013314967

Paper	Title	A faster algorithm for the single source shortest path problem with few distinct positive lengths
	Author(s)	James B.Orlin, Kamesh Madduri, K.Subramani, M.Williamson
	Journal or Conference	Elsevier B.V
	Date	20 March 2009

소개

1.1 Problem Statements

논문의 내용은 소셜 네트워크의 "gossip problem"으로부터 제안되었다. gossip problem이란 무엇일까? 소셜 네트워크는 참여자(participants)들의 클러스터(cluster)라 정의 할 때, 클러스터 간intra-cluster) 거리를 1이라 하고 클러스터 내부의 거리를 l 이라 하자(여기서 l > 1인 실수를 의미한다).

gossip is threading of information about particular[1]

"gossip problem" 이란 이러한 구조를 가진 소셜 네트워크 상에서 하나의 클러스터로부터 생긴 가십(gossip)이 어떻게 하면 모든 사람들에게 빠르게 전달 될 수 있는가에 대한 문제이다.

K를 간선(edge)들의 서로 다른 길이라 할 때, 이것은 K = 2인 SSSPP(Single source shortest path problem)의 특별한 경우로 볼 수 있다. 비록 여기서는 K를 2로 언급하였지만, K는 input size와 함께 점진적으로 증가할 수 있음에 유의하여야 한다. 본 논문은 이와 같이 K의 수가 제한된 상황에서 더 뛰어난 성능을 보이는 다익스트라(Dijkstra) 알고리즘을 제안함으로써 "gossip problem"을 해결할 수 있음을 보인다.

1.2 Terminology Statements

여기서, 특별한 언급이 없으면 그래프는 방향그래프(directed graph)를 의미한다고 정한다. 그래프는 G = (V, E)로 표시하며, V는 정점(vertex)의 집합 그리고 E는 간선의 집합을 의미한다. 특히, E(v)는 정점 v로부터 나온 간선을 의미한다.

L = {l1, l2, … , lk}는 음이 아닌 서로 다른 간선의 길이의 집합을 의미한다. L은 입력의 일부로 주어지고 배열에 저장된다고 가정한다. $c_{i,j}$는 (i , j) 간선의 길이를 의미하는데, 일반적으로 $c_{i,j} \in$ 라고 한다.

각 정점에 대하여 f(v)는 source s로부터 v까지의 현재의 최단 경로의 길이를 의미한다.

다익스트라는 기능을 수행하면서 다음의 두 구조체를 유지한다.
집합 S는 시작점 s로부터 최단거리가 확정된 정점들의 집합이다. 반면에, 집합 T = V -S로, 아직 최단거리가 확정되지 않는 정점들의 집합이다.

	## 1.3 Idea concept 본 논문은 바이너리 힙(binary heap)을 이용하여 다익스트라 알고리즘의 성능을 개선하였다고 설명한다. 바이너리 힙을 이용한 구현 자체는 여느 알고리즘과 다를 것이 없으나, 한 가지 가장 큰 차이점은 하나의 바이너리 힙이 아닌 $O\left(\frac{k}{q}\right)$ 개의 바이너리 힙을 사용하였다는 것이다. 저자는 제안한 알고리즘이 K 값이 작은 상황에서 $O(m + n \cdot \log n)$의 시간복잡도를 가진 피보나치 힙(Fibonacci)보다 더 나은 성능을 보인다고 주장하였다. 또한, 설사 k 가 큰 값을 가진다 하더라도 일반적인 바이너리 힙 구현과 똑 같은 $O(m \cdot \log m)$의 성능을 보인다고 하였다. 이는 바이너리 힙의 특이한 성질 때문인데, 자세한 내용은 알고리즘 장에서 다루도록 한다.
알고리즘	## 2.1 A faster algorithm if K is permitted to grow with problem size 제안한 알고리즘 먼저 다수의 바이너리 힙을 이용하여 병목이되는 FINDMIN() 함수의 성능을 향상 시킨다. FINDMIN() 함수는 임시 정점들의 집합 T 에서 가장 작은 거리(distance)값을 가진 vertex 를 찾는 함수이다. 바이너리 힙을 이용할 경우 이에 대한 시간 복잡도는 O(1)이 된다. 아래 그림은 개선된 다익스트라 알고리즘의 스도코드이다. 아래 스도코드에서 FINDMIN()함수는 let *r* = *argmin*{ *f (t)*: 1_*t* _ *K*} 로 표현되었다. *J.B. Orlin et al. / Journal of Discrete Algorithms 8 (2010) 189–198* **Function** NEW-DIJKSTRA() 1: INITIALIZE() 2: **while** $(T \neq \emptyset)$ **do** 3: let $r = argmin\{f(t)\colon 1 \leqslant t \leqslant K\}$. 4: let $(i, j) = CurrentEdge(r)$. 5: $d(j) := d(i) + l_r$; $pred(j) := i$. 6: $S = S \cup \{j\}$; $T := T - \{j\}$. 7: **for** (each edge $(j,k) \in E(j)$) **do** 8: Add the edge (j,k) to the end of the list $E_t(S)$, where $l_t = c_{jk}$. 9: **if** (CurrentEdge$(t) = NIL$) **then** 10: CurrentEdge$(t) := (j,k)$ 11: **end if** 12: **end for** 13: **for** $(t = 1$ to $K)$ **do** 14: UPDATE(t). 15: **end for** 16: **end while** Fig1. NEW-DIJKSTRA 스도코드 제안한 알고리즘의 가장 큰 핵심은 집합 T 의 정점들을 여러 개의 바이너리 힙에 나눠 유지하는 것이다. q = $\frac{nk}{m}$ 라 할 때, 바이너리 힙은 각각 $O(q)$의 크기를 가진다. 즉, 첫 번째 바이너리 힙은 j = 1 to q 에 대한 f(j)를 저장한다. 마찬가지로, 두 번째 바이너리 힙은 j = q + 1 to 2q 정점들에 대하여 f(j)를 저장한다. 이런 식으로 모든 집합 v ∈ T인 모든 정점들의 f(v)를 바이너리 힙에 나눠 저장한다.

여러 개의 바이너리 힙을 유지하는 것이 성능의 향상으로 이뤄지는 이유는, 바이너리 힙의 독특한 특성 때문이다. 바이너리 힙(Min Heap)은 최소 값을 찾는데 O(1)의 시간 밖에 걸리지 않으므로 알고리즘은 그저 각 바이너리 힙의 최소값을 비교하여 그 중 가장 작은 값을 최소값으로ㄷ; 결정하기만 하면 된다. 최소값을 찾은 후에는 해당 힙에서 element 를 삭제 해줘야 하는데, 이 시간은 하나의 바이너리 힙을 사용했을 때보다 더 적은 $O(n \cdot \log q)$의 시간이 든다. 이는 각각의 바이너리 힙의 크기가 하나의 큰 바이너리 힙보다 크기가 적기 때문이다.

The binary heap implementation of Dijkstra's algorithm
with $O(kq)$ binary heaps of size $O(q)$ with $q = \frac{nk}{m}$

위와 같은 형식으로 구현되었을 경우 제안한 알고리즘은 다음과 같은 시간 복잡도를 가진다.

$O(m)$ if nk < 2m

$O\left(m \cdot log \frac{nk}{m}\right)$ if nk ≥ 2m

다음은 다른 다익스트라 알고리즘의 시간복잡도와 비교한 도표이다.

slower algorithm	$O(m + nk)$
Fredman and Tarjan's Fibonacci Heap 구현	$O(m + n \cdot \log n)$
Atomic Heap 구현	$O\left(m + \frac{n \cdot \log n}{log \log n}\right)$

Table1. 시간 복잡도 비교

2.2 A slower algorithm with O(m+nk)

제안한 알고리즘과 비교를 위해 참조한 다익스트라 알고리즘이다. 가장 성능의 병목이 되는 부분은 역시 FINDMIN() 부분이다. 여타 다른 다익스트라 알고리즘은 우선순위큐를 구현하여 FINDMIN()의 성능을 향상 시키지만, 이 알고리즘은 우선순위큐를 사용하지 않는다. 대신에 Et(S) = {(I,j) ∈ E : i ∈ S, $c_{i,j}$ = lt } 연결 리스트를 유지한다. 이 연결리스트는 f(v) 값을 기준으로 정점들이 정렬된다. 적은 f(v)를 가진 정점일수록 연결리스트 끝에 위치하게 된다. 결론적으로, Et(S)[0]은 집합 T 중 가장 작은 f(v)를 가진 정점이 된다. FINDMIN 함수는 O(|T|)의 시간복잡도를 가진다. 거의 모든 정점들에 대하여 FINDMIN()함수를 실행해야하므로 알고리즘 내에서 FINDMIN 함수가 가지는 최종적인 시간복잡도는 $O(N^2)$이다.

분석

3.1 Research Design

논문은 다음과 같은 기준으로 실험을 설계한 후 실행시간을 BFS 실행시간에 일반화하였다.

a) Graph topology b) problem size c) value of K d) weight distribution

다음은 graph families 별 BFS 실행시간이다. BFS 실행시간의 측정 단위는 milliseconds 이다.

Problem instance		BFS time (milliseconds)
1	Sparse random, 2M vertices, 8M edges, $C = 10000$	6430
2	Dense random, 100K vertices, 100M edges, $C = 100$	150
3	Long mesh, 2M vertices, 8M edges, $C = 100$	3260
4	Square mesh, 2M vertices, 8M edges, $C = 100$	4900
5	Small-world graph, 2M vertices, 8M edges, $C = 10000$	5440

Table 2. Graph families 별 BFS 실행시간

위와 같은 조건들을 바탕으로 논문은 기존에 참조한 알고리즘과 새로이 제안한 알고리즘을 비교하였다.

3.2 Research Analysis

1) 다음은 'sparse random graph'에 대해 k 값을 기준으로 측정한 도표이다.

K = 1 to 8 에 대해, faster 알고리즘이 더 우수한 성능을 보였다.

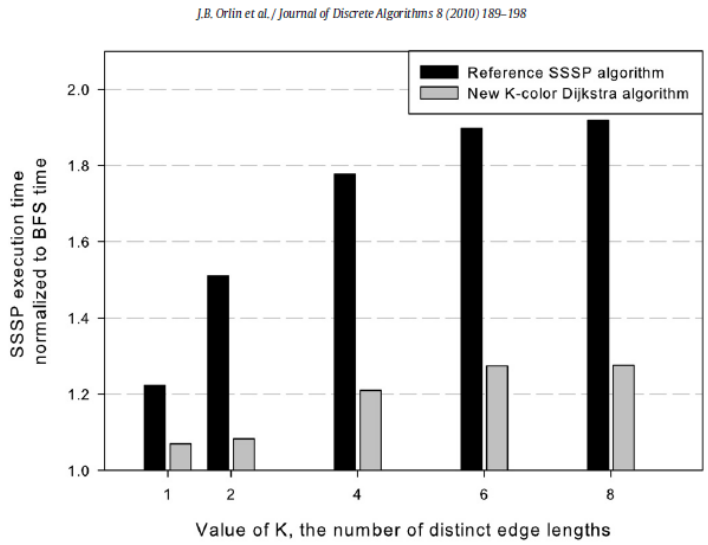


Fig2. sparse random graph – k값 기준 성능 테스트

2) 다음은 'small-world graph'에 대해 k 값을 기준으로 측정한 도표이다.

K = 4 에서만 비슷한 성능을 보였을 뿐, 그 외에는 faster 알고리즘이 더 좋은 성능을 보였다.

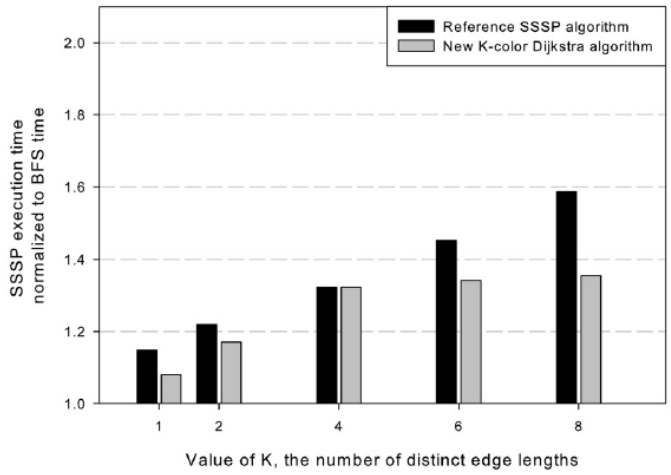


Fig3. Small world graph – k값 기준 성능 테스트

3) 다음은 'square mesh 에 대해 k 값을 기준으로 측정한 도표이다.

K = 1,2 에서만 비슷한 성능을 보였을 뿐, 그 외에는 faster 알고리즘이 더 좋은 성능을 보였다.

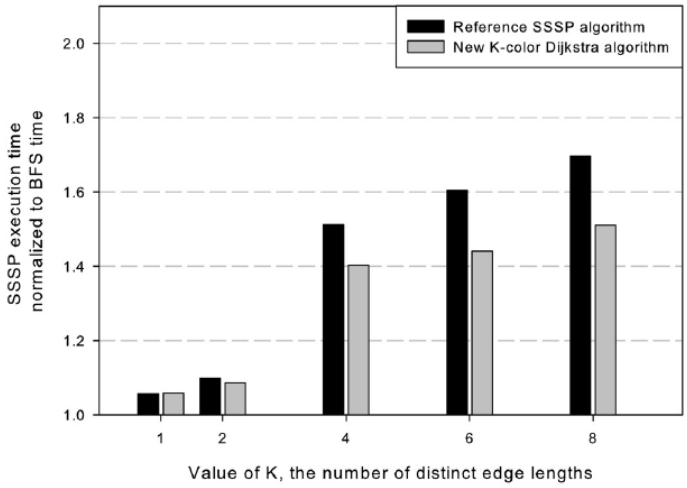


Fig4. Sqaure Mesh – k값 기준 성능 테스트

* 전반적인 실험에서 faster 알고리즘이 더 좋은 성능을 보였다. 모든 도표에 대한 설명은 생략하도록 한다.

설계 및 구현

*User function description (Or Output description) 등을 포함하여 자유롭게 서술

4.1 Faster algorithm Description

1)Design Descritiption

- 먼저 그래프는 인접 리스트로 구현하였다. 약 100만개 가량의 정점을 입력으로 받기 때문에, 인접행렬로는 메모리의 한계가 있었다. 그래프의

연결은 rand함수를 이용하였다.

- 다음은 그래프를 연결하는 핵심 코드이다. Rand 함수를 통해 L 집합의 임의의 길이를 선택한다.

```
adj[i].push_back(make_pair(j, L[rand() % k]));
```

- k*q개의 바이너리 힙을 만든 후, 다수의 바이너리 힙을 유지 및 관리하는 함수를 만들었다 – empty, push, top

- 다익스트라 알고리즘은 다음과 같이 진행된다.
  먼저 다수의 바이너리 힙을 조사하여 가장 작은 정점을 꺼낸다.
  해당 정점과 인접한 정점들을 조사하여, 더 짧은 경로를 발견할 수 있는지 확인하고 그렇다면 최단경로를 갱신한다. 또한, 그 정점을 바이너리 힙에 삽입한다.

2) Function Description

>다수의 바이너리 힙 유지 및 관리

```
void push(int _dist, int _num);
```

- 삽입하려는 vertex의 번호를 통해 어떤 바이너리 힙(우선순위큐)에 삽입되야 하는지 알 수 있다.

- vertex number/ q의 몫이 삽입 되어야할 바이너리 힙의 번호이다.

- 0부터 q-1 까지의 vertex 번호는 q로 나눴을 때 몫이 0이므로, 0번째 바이너리 힙에 저장된다. 마찬가지로, q부터 2q-1 까지의 vertex 번호는 q로 나눴을 때 몫이 1이므로, 1번째 바이너리 힙에 저장된다.

```
int pq_num = _num / q;
pq[pq_num].push(t);
```

```
bool top(vertex& vtx);
```

- 먼저, 비어있지 않은 바이너리 힙의 min 값을 모두 꺼내 비교한다.
- 가장 작은 값을 min_dist, 그에 해당하는 바이너리 힙의 번호를 min_idx로 정한다.
- 그 후, min_dist와 min_idx를 반환하여 가장 작은 f(v)를 가지는 v를 알 수 있도록 한다.
- 마지막으로 꺼낸 바이너리힙에서 해당 vertex를 pop(삭제)한다.

```
for (int i = 0; i < k*q; i++)
{
        if (pq[i].empty()) continue;
```

```
                int dist = (pq[i].top()).dist;
                if (min_dist > dist)
                {
                        min_dist = dist;
                        min_idx = i;
                }
        }
```

```
bool empty();
```

empty()함수는 모든 바이너리 힙이 empty일 경우 true를 반환하도록 구현하였다.

```
        for (int i = 0; i < k*q; i++)
        {
                if (!(pq[i].empty()))
                        return false;
        }
```

>다익스트라 알고리즘 함수

```
vector<int> dijkstra(int src);
```

- 먼저 다수의 바이너리 힙을 조사하여 가장 작은 정점을 꺼낸다.
- 해당 정점과 인접한 정점들을 조사하여, 더 짧은 경로를 발견할 수 있는지 확인하고 그렇다면 최단경로를 갱신한다.

- 또한, 그 정점을 바이너리 힙에 삽입한다.

```
        while (!empty()) {

                vertex t;
                if (top(t) == false)     break;
                int cost = t.dist;
                int here = t.number;

               // 만약 지금 꺼낸 것보다 더 짧은 경로를 알고 있다면 지금 꺼낸 것을
               무시한다.
                if (dist[here] < cost) continue;

                // 인접한 정점들을 모두 검사한다.
                for (int i = 0; i < (int)adj[here].size(); ++i) {

                        int there = adj[here][i].first;
                        int nextDist = cost + adj[here][i].second;
                        // 더 짧은 경로를 발견하면, dist[]를 갱신하고 우선순위
                        큐에 넣는다.
                        if (dist[there] > nextDist) {
                                dist[there] = nextDist;
                                push(nextDist, there);
                        }

                }
```

```
        }
```

4.2 Slower algorithm Description

1)Design Descritiption

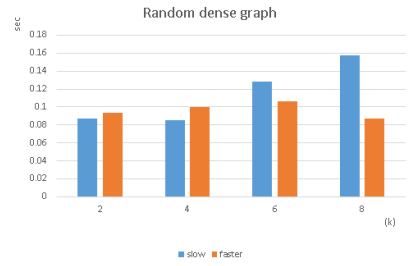
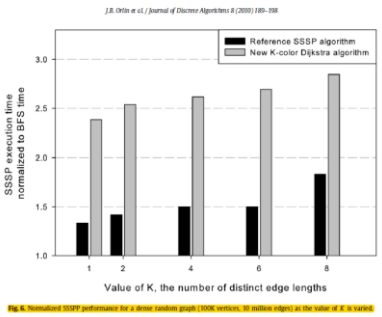
- 그래프는 faster algorithm과 마찬가지로 인접리스트로 구현하였고, 그래프 생성과정은 동일하다.
- 아직 최단 경로가 확정되지 않은 정점들의 집합 T를 나타내는 벡터를 유지한다.
- 매 반복시마다 T 벡터를 정렬한 후 가장 작은 f(v)를 가지는 정점을 꺼낸다.
- 해당 정점과 인접한 정점들을 조사하여, 더 짧은 경로를 발견할 수 있는지 확인하고 그렇다면 최단경로를 갱신한다.
- 또한, 그 정점을 T 벡터에 삽입한다.

2) Function Description

```
vector<int> dijkstra(int src);
```

- 매 반복시마다 T 벡터를 정렬한 후 가장 작은 f(v)를 가지는 정점을 꺼낸다.
- 해당 정점과 인접한 정점들을 조사하여, 더 짧은 경로를 발견할 수 있는지 확인하고 그렇다면 최단경로를 갱신한다.
- 또한, 그 정점을 T 벡터에 삽입한다.

```
        while (!t.empty()) {

                sort(t.begin(), t.end());// t[0]은 가장 작은 값
                int cost = t[0].first;
                int here = t[0].second;

                t.erase(t.begin() + 0);

                        // 만약 지금 꺼낸 것보다 더 짧은 경로를 알고 있다면 지금
                 꺼낸 것을 무시한다.
                if (dist[here] < cost) continue;

                // 인접한 정점들을 모두 검사한다.
                for (int i = 0; i < (int)adj[here].size(); ++i) {

                        int there = adj[here][i].first;           // 정점 번호
                        int nextDist = cost + adj[here][i].second;
                        // 더 짧은 경로를 발견하면, dist[]를 갱신하고 우선순위
                        큐에 넣는다.
                        if (dist[there] > nextDist) {
                                dist[there] = nextDist;
                                t.push_back(make_pair(nextDist, there));
                        }

                }
```

	}
논문의 결과 및 평가결과	5.1 Comparision 실제 측정 데이터와 논문 내 그래프를 비교해본다. 단, 논문 내 데이터는 BFS 실행 시간으로 표현 되었으나 검증에서는 CPU 시간으로 데이터를 측정하였다. 1) TEST #1 - Random dense graph - 100K vertices, 10 million edges  도표1. 실제 측정 데이터  도표2. 논문 내 데이터 *comparision : faster가 더 나은 성능을 보이는 것은 확실하나 논문 내 데이터처럼 확연한 성능의 차이를 보이지는 않는다. BFS Time이 아닌 cpu 시간으로 측정하였기 때문에 그래프 간 차이가 존재하는 것 같다. 2) TEST #2 - Random sparse graph - 4M vertices, 16M edges

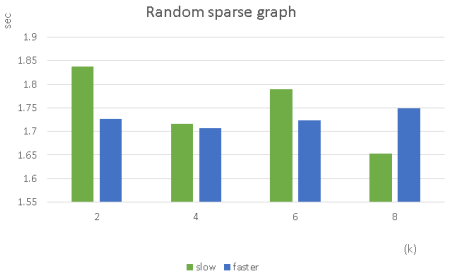


도표3. 실제 측정 데이터

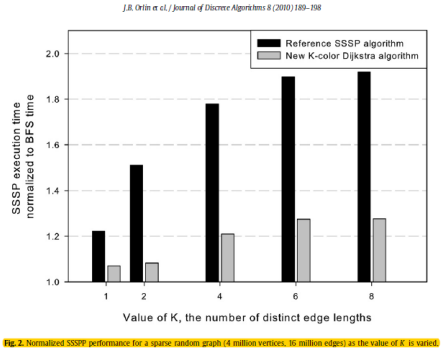
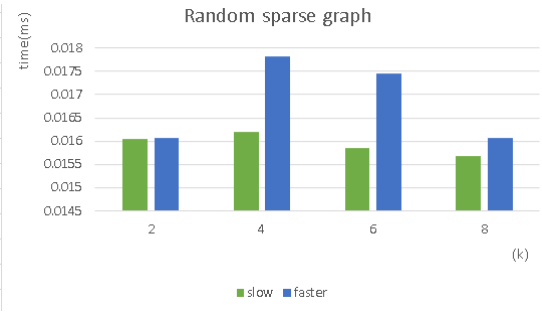


도표4. 논문 내 데이터

*comparision : 확실한 것은 제안한 알고리즘이 더 나은 성능을 보인 다는 것이다.

3) TEST #3

- Random sparse graph
- 80 vertices, 160 edges

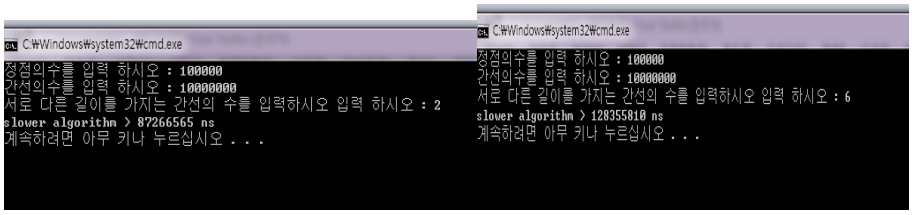


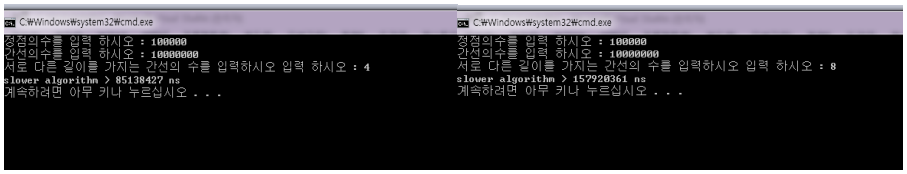
*comparision : 이 경우 오히려 개선한 알고리즘의 성능이 더 안 좋게 나왔다.

<시간 측정 함수>

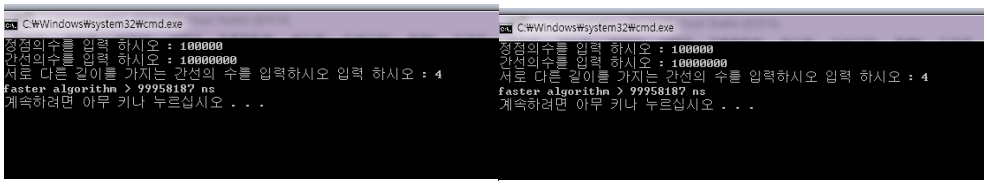
```
auto start = chrono::high_resolution_clock::now();
dijkstra(0);
auto end = chrono::high_resolution_clock::now();
cout << "faster algorithm > " << (end - start).count() << " ns" << endl;
```

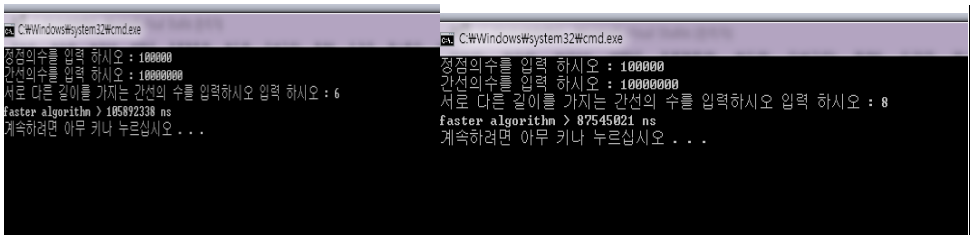
<slower 출력화면 – TEST #1>





<faster 출력화면 – TEST #1>





5.2 Essay

세 차례의 테스트를 진행한 결과 논문의 데이터와 유사한 그래프 형태가 나왔다. 확실한 것은 제안한 알고리즘이 더 나은 성능을 제공한다는 것이다.

하지만, 한 가지 함정인 부분이 있다. 대부분의 다익스트라 알고리즘이 성능적인 문제로 우선순위 큐를 이용하나, 실험에서는 우선순위 큐를 이용하지 않는 알고리즘을 비교 대상으로 삼았다는 것이다. 비록 비교 대상인 알고리즘이 정렬을 이용하여 다익스트라 알고리즘을 구현하였으나, 반복문마다 f(v)가 최소인 정점을 반환함을 보장한다는 것은 알고리즘 선택의 타당성을 어느 정도 뒷받침해준다. 즉, 원리상으로는 우선순위 큐와 같다는 것이다. 하지만, 매 회 O(T) ≈ O(V) 시간복잡도의 정렬을 해야하는 점을 비춰볼 때 다른 우선순위 큐 구현 알고리즘보다는 성능이 떨어지는 것이 사실이다. 정말로 실효성 있는 실험을 위해서는 피보나치 힙 혹은 아토믹 힙을 통해 구현한 알고리즘과 비교하는 것이 타당하다고 생각한다. 아마도, 제안한 알고리즘의 성능 데이터를 과장하기 위해 적당한 알고리즘을 선택하지 않았나 추측해 본다.

하지만, 다수의 바이너리 힙을 이용한다는 아이디어와 실험 데이터를 정밀하게 분석한 부분은 우수했다고 생각한다. 덕분에, 논문을 분석하면서 개인적으로도 좋은 공부가 되었던 것 같다.

소스코드

6.1 Configuration Description

윈도우 c+11 이상 환경에서 실행 가능합니다. 정점의 수, 간선의 수, K의 수 (2,4,6,8..) 을 입력하면 rand 함수에 의해 적절히 그래프가 형성 됩니다.

6.2 faster algorithm Source code

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <algorithm>
using namespace std;
#define INF INT_MAX

struct vertex
{
public:
        int dist;
        int number;

};
```

```
struct Comp
{
	bool operator()(vertex v1, vertex v2)
	{
		return v1.dist > v2.dist;// 낮은 순으로 뽑는다
	}
};

vector<priority_queue<vertex, vector<vertex>, Comp> > pq;
vector<vector<pair<int, int> > > adj;	// 그래프의 인접 리스트. (연결된 정점 번호, 간선 가중치) 쌍을 담는다.
vector<int> L;

int v, e, k, q;

// -삽입하려는 vertex의 번호를 통해 어떤 바이너리 힙(우선순위큐)에 삽입되야 하는지 알 수 있다.

//- vertex number / q의 몫이 삽입 되어야할 바이너리 힙의 번호이다.

//- 0부터 q - 1 까지의 vertex 번호는 q로 나눴을 때 몫이 0이므로, 0번째 바이너리 힙에 저장된다.마찬가지로, q부터 2q - 1 까지의 vertex 번호는 q로 나눴을 때 몫이 1이므로, 1번째 바이너리 힙에 저장된다.

void push(int _dist, int _num)
{
	vertex t;
	t.dist = _dist;  t.number = _num;

	int pq_num = _num / q;
	pq[pq_num].push(t);
}

//-먼저, 비어있지 않은 바이너리 힙의 min 값을 모두 꺼내 비교한다.
//- 가장 작은 값을 min_dist, 그에 해당하는 바이너리 힙의 번호를 min_idx로 정한다.
//- 그 후, min_dist와 min_idx를 반환하여 가장 작은 f(v)를 가지는 v를 알 수 있도록 한다.
//- 마지막으로 꺼낸 바이너리힙에서 해당 vertex를 pop(삭제)한다.


bool top(vertex& vtx)
{
	int min_idx = -1;
	int min_dist = INF;

	for (int i = 0; i < k*q; i++)
	{
		if (pq[i].empty()) continue;	// 비어있는 바이너리 힙은 제외한다.

		int dist = (pq[i].top()).dist;
		if (min_dist > dist)
		{
			min_dist = dist;
			min_idx = i;
```

```
                }
        }

        if (min_idx != -1)
        {
                vtx = pq[min_idx].top();
                pq[min_idx].pop();

                return true;
        }

        else
        {
                return false;    // false 반환은 우선순위 큐가 비었다는 것을 의미
        }
}


// empty()함수는 모든 바이너리 힙이 empty일 경우 true를 반환하도록 구현하였다.
bool empty()
{
        for (int i = 0; i < k*q; i++)
        {
                if (!(pq[i].empty()))
                        return false;
        }

        return true;
}

vector<int> dijkstra(int src) {

        vector<int> dist(v, INF);        // dist는 최단 경로 비용 배열이다.

        pq.resize(k*q);          // k*q는 우선순위 큐의 개수
        dist[src] = 0;

        push(0, src);

        while (!empty()) {

                vertex t;
                if (top(t) == false)     break;
                int cost = t.dist;
                int here = t.number;

                // 만약 지금 꺼낸 것보다 더 짧은 경로를 알고 있다면 지금 꺼낸
것을 무시한다.
                if (dist[here] < cost) continue;

                // 인접한 정점들을 모두 검사한다.
                for (int i = 0; i < (int)adj[here].size(); ++i) {

                        int there = adj[here][i].first;
                        int nextDist = cost + adj[here][i].second;
```

```
                        // 더 짧은 경로를 발견하면, dist[]를 갱신하고 우선순위
큐에 넣는다.
                        if (dist[there] > nextDist) {
                                dist[there] = nextDist;
                                push(nextDist, there);
                        }

                }
        }

        return dist;
}

int main(void)
{

        // 1. 정점의 수(v), 간선의 수(E), K를 입력받고 L을 만든다. RAND 함수 이용

        // 2. 그래프의 인접 행렬을 초기화 한다. : INF - 연결안됨, 0 - i=i, val -
연결됨
        //

        //
        // 3.  RAND 함수를 이용하여 간선을 할당 한다. 중요한 것은 L의 값을
이용해야 한다는 것이다.


        srand((unsigned int)time(NULL));

        cout << "정점의수를 입력 하시오 : ";
        cin >> v;

        cout << "간선의수를 입력 하시오 : ";
        cin >> e;


        cout << "서로 다른 길이를 가지는 간선의 수를 입력하시오 입력 하시오 : ";
        cin >> k;

        q = (v*k) / e;

        L.reserve(k);
        for (int i = 0; i < k; i++)
        {
                L.push_back((rand() + 1) % INF);
        }

        adj.resize(v);

        int t = e;

        for (int i = 0; i < v - 1; i++)
        {
                int cnt = e / v;
```

```
                for (int j = 0; j < cnt; j++)
                {
                        adj[i].push_back(make_pair(j, L[rand() % k]));
                }
                t -= cnt;
        }

        int tt = t;
        for (int j = 0; j < tt; j++)
        {
                adj[v - 1].push_back(make_pair(j, L[rand() % k]));
                t--;
        }

        if (t != 0) return 0;

        auto start = chrono::high_resolution_clock::now();

        dijkstra(0);

        auto end = chrono::high_resolution_clock::now();

        cout << "faster algorithm > " << (end - start).count() << " ns" << endl;
        return 0;
}
```

6.3 Slower algorithm Source code

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <chrono>
#include <algorithm>
using namespace std;
#define INF INT_MAX



vector<vector<pair<int, int> > > adj;     // 그래프의 인접 리스트. (연결된 정점
번호, 간선 가중치) 쌍을 담는다.
vector<int> L;

int v, e, k, q;

/*
- 매 반복시마다 T 벡터를 정렬한 후 가장 작은 f(v)를 가지는 정점을 꺼낸다.
- 해당 정점과 인접한 정점들을 조사하여, 더 짧은 경로를 발견할 수 있는지 확인하고
그렇다면 최단경로를 갱신한다.
- 또한, 그 정점을 T 벡터에 삽입한다.
```

```
*/
vector<int> dijkstra(int src) {

	vector<int> dist(v, INF);
	vector <pair<int, int> > t;	// first : dist, second : num

	dist[src] = 0;
	t.push_back(make_pair(src, 0));


	while (!t.empty()) {

		sort(t.begin(), t.end());// t[0]은 가장 작은 값
		int cost = t[0].first;
		int here = t[0].second;

		t.erase(t.begin() + 0);

		// 만약 지금 꺼낸 것보다 더 짧은 경로를 알고 있다면 지금 꺼낸
것을 무시한다.
		if (dist[here] < cost) continue;

		// 인접한 정점들을 모두 검사한다.
		for (int i = 0; i < (int)adj[here].size(); ++i) {

			int there = adj[here][i].first;		// 정점 번호
			int nextDist = cost + adj[here][i].second;
			// 더 짧은 경로를 발견하면, dist[]를 갱신하고 우선순위
큐에 넣는다.
			if (dist[there] > nextDist) {
				dist[there] = nextDist;
				t.push_back(make_pair(nextDist, there));
			}

		}
	}

	return dist;
}

int main(void)
{

	// 1. 정점의 수(v), 간선의 수(E), K를 입력받고 L을 만든다. RAND 함수 이용

	// 2. 그래프의 인접 행렬을 초기화 한다. : INF - 연결안됨, 0 - i=i, val -
연결됨
	//

	//
	// 3.  RAND 함수를 이용하여 간선을 할당 한다. 중요한 것은 L의 값을
이용해야 한다는 것이다.
```

```
	srand((unsigned int)time(NULL));

	cout << "정점의수를 입력 하시오 : ";
	cin >> v;

	cout << "간선의수를 입력 하시오 : ";
	cin >> e;


	cout << "서로 다른 길이를 가지는 간선의 수를 입력하시오 입력 하시오 : ";
	cin >> k;

	q = (v*k) / e;

	L.reserve(k);
	for (int i = 0; i < k; i++)
	{
		L.push_back((rand() + 1) % INF);
	}

	adj.resize(v);

	int t = e;

	for (int i = 0; i < v - 1; i++)
	{
		int cnt = e / v;

		for (int j = 0; j < cnt; j++)
		{
			adj[i].push_back(make_pair(j, L[rand() % k]));
		}
		t -= cnt;
	}

	int tt = t;
	for (int j = 0; j < tt; j++)
	{
		adj[v - 1].push_back(make_pair(j, L[rand() % k]));
		t--;
	}

	if (t != 0) return 0;

	auto start = chrono::high_resolution_clock::now();

	dijkstra(0);

	auto end = chrono::high_resolution_clock::now();

	cout << "slower algorithm > " << (end - start).count() << " ns" << endl;
	return 0;
}
```

참고문헌	[1] GOSSIP: IDENTIFYING CENTRAL INDIVIDUALS IN A SOCIAL

	NETWORK.pdf [2] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM 34 (3) (1987) 596–615. [3] INTRODUCTION TO ALGORITHMS – 3rd edit
의견	처음에는 이해하기 어려운 내용이 많았으나, 프로젝트를 진행하면서 개인적으로 좋은 공부가 되었던 것 같습니다. 나중에 논문을 접할 때 좋은 밑거름이 될 것 같습니다.

* 12 주차 ~ 14 주차:

- 선정 논문내의 알고리즘 이해 및 분석
- 입.출력 모듈 구현
- 입력, 출력 및 각종 기능의 효율성 향상을 위한 설계 및 구현
- 논문 내의 입력(또는 각종 입력)에 대한 실행 및 시스템 검증

* 15 주차: 설계 보고서 제출 및 구현물 발표

- 선정 논문의 효용성, 복잡도, 완성도, 실행효율성 등을 고려하여 구현물 평가
- 리포트 내용 평가

평가방법

1. 대상 논문의 이해/분석 및 알고리즘의 효용성과 복잡도 (40%)
2. 설계의 접근 방법 및 구현 완성도 (40%)
3. 설계보고서 평가 (20%)