**컴퓨터 구조론2 최종 과제**

컴퓨터공학과 201211355 손지웅

**1. 알고리즘**

초기 Vertex배열에 시작 노드를 추가하고 Vertex 배열에서 가장 작은 거리를 가진 Vertex를 찾고 각 노드의 최단 거리 배열에서의 같은 인덱스에 있는 값을 비교하여 최단거리 배열 값이 더 작 다면 연산하지 않고 같거나 크다면 인접 정점을 돌면서 최단거리 배열 값보다 작은 값으로 갱신하는 알고리즘 즉, 평범한 다익스트라 알고리즘을 이용하였습니다.

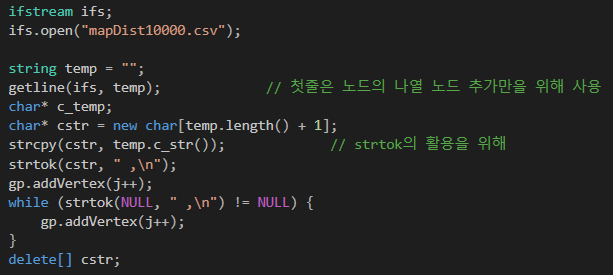
처음에는 우선순위 큐 나아가 튜닝을 하여 빠른 성능의 push\_heap, pop\_heap(배열을 사용하고 메모리 재할당을 없앰)을 하여 Vertex배열의 최소 값을 구하였는데 이는 노드의 개수가 적을 때는 매우 빠른 성능 O(Elogn) 을 내었지만 병렬화를 효율적으로 이용하는 방법을 찾지 못했습니다. 이에 omp parallel for reduction을 통한 최소 값을 찾는 것을 진행하였는데 scale이 커질수록 훨씬 빠른 속도를 보였습니다.

기본 코드의 틀은 <http://hsp1116.tistory.com/42> 에서 가져와서 변형시켰습니다.

STL들을 접근 속도를 위해 거의 배열로 변형 하였고 간선은 모든 간선을 다 돌아야 하기 때문에 추가된 만큼만 반복할 수 있게 vector로 하였고 pop대신 인덱스를 줄이는 방향으로 반복문안에서 불필요하게 반복되는 연산은 밖으로 빼는 방식으로 최고의 성능을 내보았습니다.

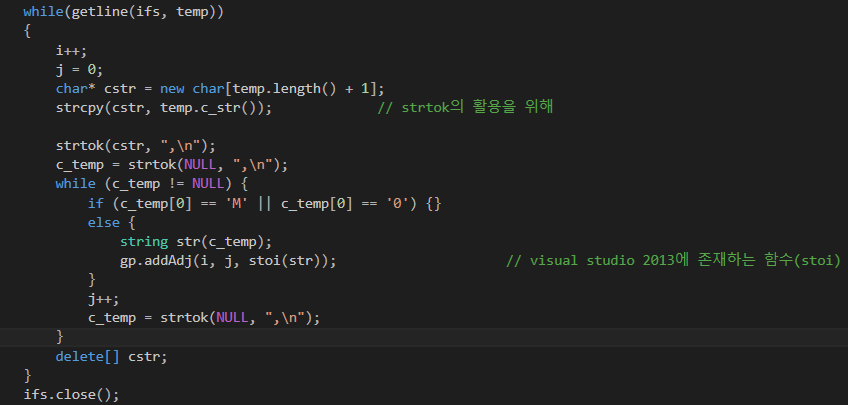
**2. 구현**

**- 파일 읽는 부분**



C++ 언어(C++14)로 구현하였습니다. C++ 스타일로 ifstream을 통해 인풋 파일스트림을 열고 getline을 통해 한 줄 단위로 csv파일 (mapDist10000.csv(10000개 노드), 파일 이름 고정)을 읽었습니다. csv파일의 첫 번째 라인이 Vertex의 나열로 되어있기 때문에 해당 부분을 comma(‘ , ’) 단위로(csv 파일은 ,가 포함되어 있음) 토큰을 분리하여 토큰이 생길 때 마다 vertex를 추가하여 주었습니다.

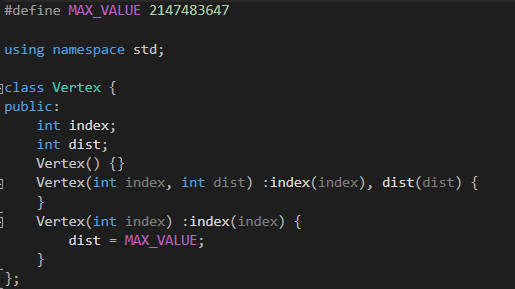
C++스타일로 구현하였지만 토큰을 분리할 때는 comma단위로 토큰 분리를 편리하게 하기 위해 strtok를 사용하였습니다. strtok사용을 위해서는 char\*형태로의 변환이 필요하므로 char\* 타입을 동적 할당하고 문자열 복사를 한 뒤에 토큰 분리를 진행하였습니다. 혹시 모를 파일의 오류를 대비해 comma, enter, space 전부 delimiter로 넣어 토큰을 분리하였습니다.



Vertex의 나열 이후부터 각 Vertex 간 거리가 나오기 때문에 i, j변수를 두어 i를 통해 열의 Vertex의 번호를 j를 통해 행의 Vertex의 번호를 하여 시작 Vertex부터 전체 Vertex를 j를 통해 돌며 간선을 추가해줍니다. 이를 위해 동일하게 strtok를 통해 토큰분리를 합니다. 그때 MAX 혹은 0이라는 값이 나오면 넘어가고 실제 간선거리만을 추가해줍니다. 이를 위해 배열보다는 STL 벡터(vector)로 간선(adj)을 위한 자료구조로 이용하였습니다. 배열에서는 중간 값이 비어도(MAX, 0으로 인한) 모든 인덱스를 돌며 확인하는 번거로움이 있겠지만 벡터에서는 벡터의 크기만을 구해서 그 크기만큼의 인덱스를 돌면 되는 이점이 있습니다.

string형을 int형으로 변환하기 위해 stoi함수를 썼는데 visual studio 2013 환경에서는 당연히 지원하는 함수라 실행환경이 visual studio 2013이라면 작동할 것입니다.

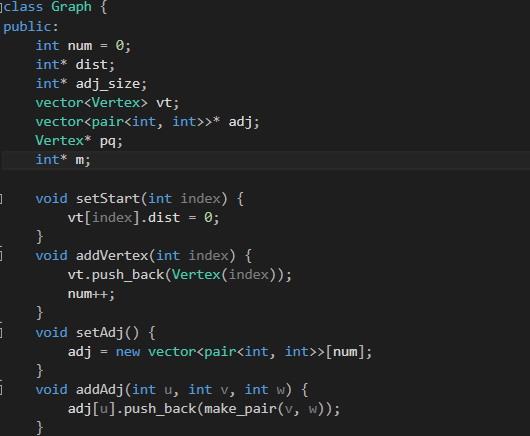
**- Dijkstra (Vertex)**



Dijkstra 알고리즘 구현을 위한 Vertex 입니다. Performance를 생각하여 std::pair 혹은 struct, class 중에 어떤 것을 쓸까 하다가 차이가 없고 이후 생성자를 통해 편리하게 이용할 부분이 있어 클래스로 하였습니다.

자신의 index 번호와 이전 노드로부터의 거리를 가지게 되며 초기에는 MAX\_VALUE(int 최대값)을 가지게 됩니다. 기본 생성자는 배열 생성을 위해 만들었습니다.

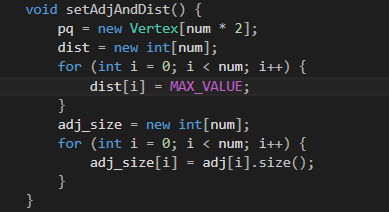
**- Dijkstra (frame)**



그래프 클래스입니다.

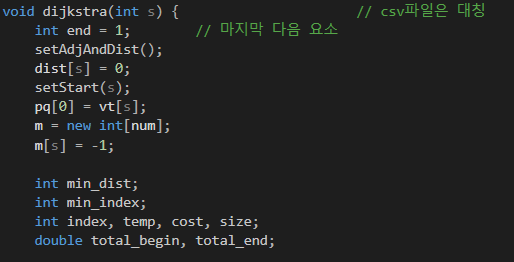
최단거리 배열 dist, 각 노드마다의 간선 수를 계산하는 adj\_size, 기본 Vertex vector vt, 간선 vector adj(도착지, 거리), Vertex 배열 pq(다익스트라에서 활용), 경로 배열 m을 선언했습니다.

main에서 사용한 것처럼 Vertex를 추가하는 부분 addVertex(), 간선을 추가하는 부분addAdj()가 존재합니다. 이후 시작 노드의 dist 값의 초기화를 하는 setStart() (굳이 필요하지는 않습니다), 더 빠른 성능을 위해 adj를 벡터가 아닌 배열로 생성하였으므로 배열의 동적 할당을 하는 setAdj()가 존재합니다.



setAdjAndDist()는 최단 거리배열들을 MAX\_VALUE로 초기화 시키고 pq를 std::vector보다 array로 선언함으로써 빠른 접근 및 인덱스 감소를 통한 제거로 빠른 성능을 냈습니다. Array의 미리 공간을 넉넉하게 동적 할당(재 할당을 없앰) 하였습니다.

**현재 csv파일은 간선이 다소 적은 형태로 생성되는 방식이지만 만약 간선 수가 엄청 많은 그래프라면 pq array에서 index를 넘어가는 문제가 생길 것입니다.** (극단적으로 한 노드가 모든 노드에 연결 된 형태 및 그런 노드가 여러 개 일 경우) **따라서 index오류가 난다면 array를 동적할당 하는 부분의 크기 num \* 2를 충분히 크게 잡아야 합니다. 이는 경로 찾는 부분 및 병렬화 부분이 아니기 때문에 문제가 생기면 크기를 변경해 주시면 감사하겠습니다.**



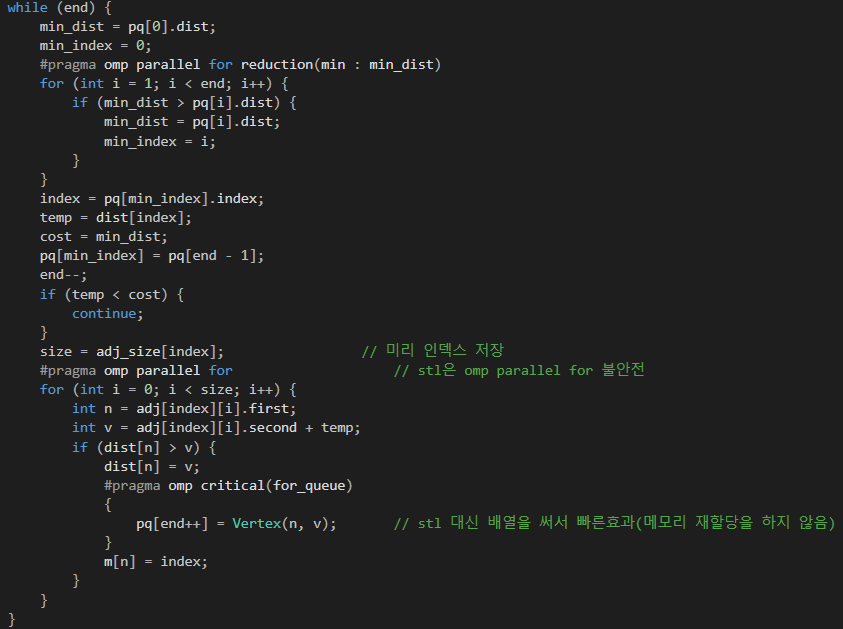
다익스트라를 들어가기 앞서 미리 설정하는 부분입니다. 우선 기존 pop 혹은 pop\_heap 연산을 안 쓰고 인덱스를 조절하는 것으로 Vertex 삭제를 대신 하기 위해 end라는 변수로 마지막 index 다음 위치를 가리키게 하였습니다.

위에서 설명한 setAdjAndDist 함수를 통한 초기화를 진행하였고 시작 인덱스의 최단거리 배열 및 Vertex의 dist값을 초기화 하여 주었고 pq 배열의 첫번째 위치에 넣어주었습니다.

또한 거리를 나타내기 위해 사용하는 int 배열인 m의 종료 위치를 m[s] = -1을 통해 넣어주었습니다.

이후 변수선언은 반복문 안에서 계속 선언하는 것보다 미리 선언하고 반복문안에서 값만 변경하는 방식이 빠를 거라 생각되어 밖으로 뺐습니다.

**- Dijkstra (Algorithm)**



실제 경로를 찾는 부분을 수행하는 부분이고 해당 부분을 시간 측정하였습니다.

While문이 돌 때마다 Vertex 배열(pq)에서 최소 거리 값을 가진 Vertex를 구하기 위해 reduction을 통한 최소 값(최소 값을 가지는 인덱스를 같이 구함)을 구하게 하였습니다. 이는 여러 번의 실험결과 최소 값을 구할 때 가장 빠른 성능을 보였습니다.

이후 Vertex의 최소 거리(이어진 간선의 최소값)와 최단거리 배열의 해당 인덱스 값과 비교하고 그에 따라 continue를 할지 계속 진행할지 정하였습니다.

속도 향상을 위해 pop연산 대신 end의 인덱스 값을 줄였습니다.

이후 해당 인덱스의 간선들의 수를 계산하는 부분에서 size = adj.size()를 진행하기보다 미리 사이즈들을 다 구해주었기 때문에 size = adj\_size[index]로 더 빠른 성능을 낼 수 있었습니다.

이후 간선배열을 사이즈만큼 탐색하며 최단거리 배열 및 Vertex배열, 경로를 저장하는 맵 배열을 갱신하는 부분을 병렬화 하였습니다. 출발지에서 도착지까지의 중복되는 경로가 나오지 않으므로 도착지는 다 다르기 때문에 스레드에 할당하여 각각 연산을 하여도 동시성이 발생하지 않기 때문입니다.

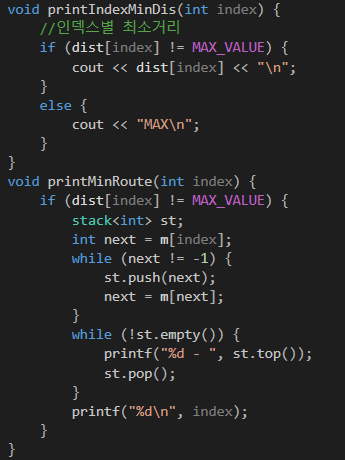
간선배열의 첫번째 인자(도착지)를 n에 할당하고 두번째 인자(index로부터 거리)와 index까지의 최단거리의 합을 v에 할당하였습니다.

이후 if문을 통해 현재 n까지의 최단거리와 index를 거쳐 n까지 가는 최단거리를 비교하여 최소값으로 dist[n]을 갱신하여 줍니다. 만약 갱신이 진행된다면 Vertex배열 pq에 현재 갱신되는 인덱스 n과 거리 v의 Vertex를 추가하고 end값을 증가시켜 줍니다.

경로 배열에 또한 넣어줍니다. 이때 end값의 갱신 및 pq배열의 추가 과정에서 end값의 동시성이 생길 수 있으므로 parallel omp critical()을 통해 동시성을 제거하였습니다.

이때 critical 영역이 단순 end값의 증가 및 배열요소 변경이므로 큰 오버헤드가 생기지는 않습니다.

**- Dijkstra (Print)**



경로를 다 계산하고 최단 거리 및 경로를 출력하는 부분입니다.

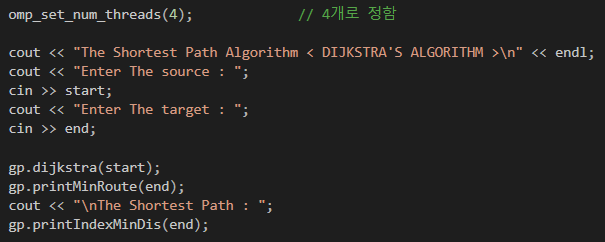
**printfIndexMinsDis(int index)**

최단 거리는 최단거리 배열에 계속 갱신하며 진행하였으므로 경로 계산부분 이 끝난 뒤 최단거리 배열에 값이 결국 최단 거리이므로 해당 인덱스의 거리만 출력해주었습니다.

**printMinRoute(int index)**

병렬화 구역에서 최단거리 배열(dist)이 갱신 될 때마다 경로 배열(m) 또한 요소를 추가해 주었습니다. 배열 값이 이전 인덱스이기 때문에 이전 인덱스를 계속 따라간 값들을 역으로 출력할 필요가 있었습니다. 따라서 스택에 담고 이를 pop하며 역으로 출력하는 방식을 취했습니다. -1 값일 땐 종료 인덱스이므로 스택에 담지 않기 때문에 마지막에 출력해주었습니다.

**- Main**



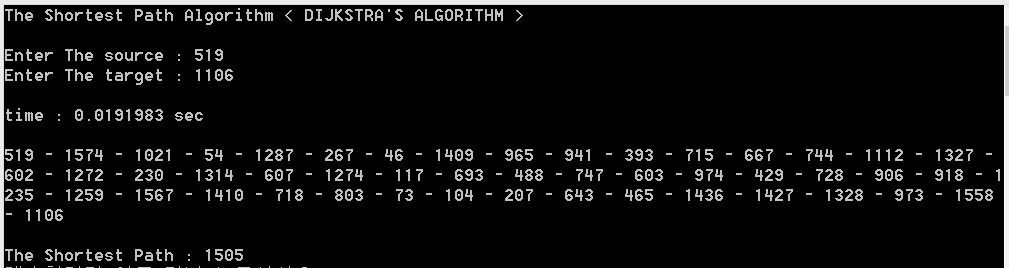
Main의 남은 부분입니다. 스레드 4개를 설정하고 시작점과 끝점을 받고 다익스트라를 실행하고 최단거리 및 경로를 출력합니다.

**3. 성능**

성능측정 환경 **CPU : Intel® Core™ i5-5200U CPU @ 2.20GHz 듀얼코어 Memory : 8GB IDE : VisualStudio 2015, Intel Parallel Studio 17, OS : Windows 8.1 K 64bit** 에서 성능 측정을 진행하였습니다.과제를 평가하실 때 환경이 **Intel i7-4790 @ 3.60GHz에 쿼드 코어, 16GB메모리를** 가지고 있어서 **현재 저의 성능(시간측정)보다 훨씬 좋게(빠르게) 나올 것으로 예상합니다.**

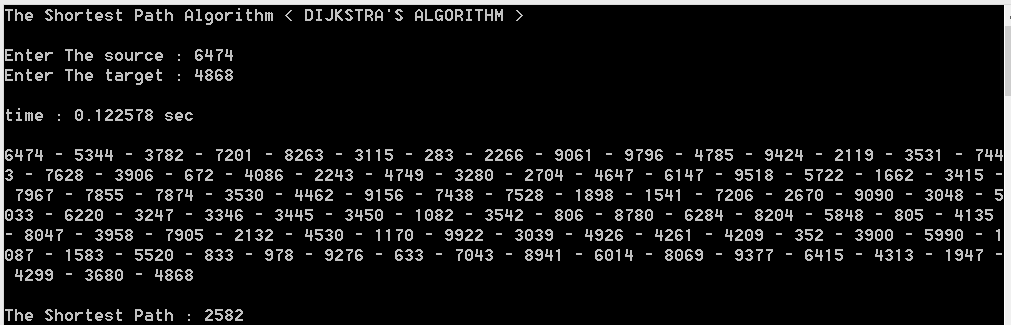
노드 1600개에 대해 첨부하신 결과물과 비교하는 정확도 및 성능 테스트, 노드 10000개에 대해 성능테스트를 진행하였습니다. 상당히 많이 진행하였지만 대표적으로 4개의 결과만 첨부하였습니다. 속도 측정은 **omp\_get\_wtime()**을 이용 하였습니다. (**측정 함수마다 다른 결과를 나타냅니다**.)

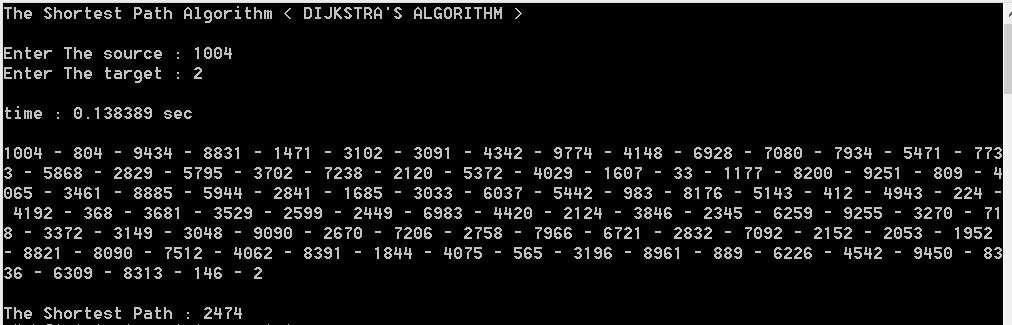
<노드 1500개 성능 및 정확도 테스트>

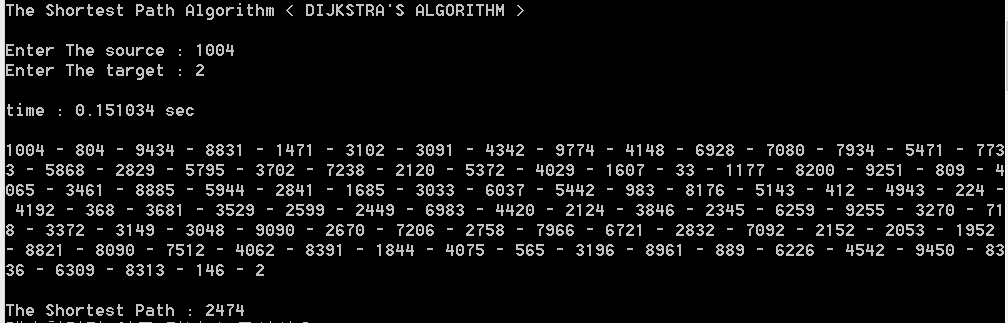


보내주신 캡쳐 화면과 경로 및 최단거리가 동일하게 출력되었습니다. 1600개에 대해서 0.0191983초의 속도를 보였습니다.

<노드 10000개 성능테스트>







노드 10000개에 임의의 출발지와 도착지를 넣어봤습니다. 각각 0.122578, 0.138369, 0.151034초의 속도를 보였습니다.

마지막 2개의 결과를 봤을 때 같은 인풋에 대해 다른 속도를 보였는데 이는 수행될 때 다른 프로세스가 돌게 되면 더욱 느린 결과를 냈고 Reduction의 수행이 일정하지 않은 것으로 보였습니다.

그 결과 10000개 노드에 대해 같은 인풋을 넣었을 때 주로 0.1xx값이 많았지만 0.11 ~ 0.24까지 다양한 속도를 보였습니다. 성능 측정시에도 여러 번 측정하여 최소 값 혹은 평균을(지나치게 큰 값 및 작은 값을 제외한) 택하는 방향이면 좋을 것 같습니다.