Algorithm Assignment#4

German language and literature 2016130927

Park Jun Yeong

Ⅰ. How can we compute c(u, v) for all possible pairs of nodes u and v in G.

An algebraic structure C = (S, +, ∙, \*, 0, 1), closed semiring is correspondent to this path-finding problem: the sum of the labels of all the paths between node u and node v. So, each label of the edge, say *l*, the label between nodes belongs to the S. Since the labels of a path of zero length is defined as 1 and the sum over an empty set of paths as 0, these properties are derived:

For all label *l* ∈ S (note that 0, 1 ∈ S)

*l* + 0 = 0 + *l = l* : 0 is a neutral element for sum.

*l* ∙ 1 = *l* ∙ 1 = *l* : 1 is a neutral element for product.

*l ∙* 0 = 0 ∙ *l =* 0 : 0 is an annihilator for product.

and then from these binary operations + and ∙:

+ : associative, commutative, idempotent(*l + l = l)*

∙ : associative, distributive over sum operation

and for \*, closure operation:

*l*\* = 1 + *l*1 + *l*2 + … (*l*n denotes that *l*∙*l*∙*l*∙…*l* product of *l* n times*)*

To compute c(u, v), labels of paths between nodes u and v in G are required. At this moment, you can observe these situations as following:

(1). **No paths from node u to node v and from node v to node u**

: If there is no connected edges from node u to node v or from node v to node u (even when all of possible intermediate edges are used), then all of possible label to be observed is 0 - it is defined as the sum of an empty set of paths. In this case, c(u, v) is easily computed as 0.

(2). **Path(s) from node u to node v and from node v to node u**

: A path refers to possible connection from node u to node v via connected edges: note that this graph G is a directed graph. Then, you would say that path comes out when you start from one edge (from node u), via zero or more intermediate edges and finally arrive at the end node v.

If there exists an edge from u to v directly, then it could be a path easily. Then its label becomes just its edge’s label. If not, you should check which nodes could be taken as an intermediate node: from node u via that a certain new node w(it is pointed by node u and points to node v), i. e., by the path from u to w, you could find a path from u to v. (this process is same as in transitive closure check): a certain path could be used as an intermediate path.

In all of possible paths from node u to v, you can imagine the set of possible selection of edge(s): these selections are distinct (there is no same route, only one route, in which some edges are selected so that).

Even though you could select a path of zero length as an intermediate edge as many times as possible, the label of that is defined as 1 so that the product in that path cannot change. Note that 1 is a neutral element for product operation.

From this set, each label of path could be easily counted with product operation. Then, these labels could be also summed to c(u, v) with sum operation. This is how to compute labels of all of possible paths from node u to v.

Likewise this computation, the sum of labels of all of possible paths from node v to u is also counted (by definition you have to find out all of paths between u and v, not from u to v and this is a directed graph).

(\*if you mean that ‘between’ is just from node u to node v, this count should be skipped. If not, it is summed.)

Therefore, we can compute c(u, v) for all possible pairs of nodes u and v in G with product and sum operations in the closed semiring.

Ⅱ. Explain 2-coloring problem

(1). **2-coloring problem as a bipartite graph**: 2-coloring problem is a decision problem, a problem to decide whether in a certain undirected graph G = (V, E), each node can be colored by one of two colors given that colors of two adjacent nodes are different: Since then, if possible, a set of nodes V = R∪B (Red and Blue) such that R∩B = and all edges e∈E are such that e is (r, b) where r∈R and b∈B: That is, 2-colored graph is a bipartite graph, whose nodes V are partitioned into two disjoint set R and B such that each edge e is incident to one node r in set R and one node b in set B. therefore, there exist no adjacent nodes in a certain partitioned set. Then, how could be whether a directed graph G is a bipartite graph(=2-colored problem) found out? It is strongly related with a certain property of this bipartite graph: a undirected graph G is bipartite if and only if there is no odd cycles

(2). **Proof by contradiction**: Let an undirected graph G = (V, E) be a bipartite graph whose nodes are partitioned into sets R and B. If there is at least one cycle C between R and B, then there are two cases:

1). the length of C is *even*. Then C = {v1-v2-…-vn-v1} (n is even). let v1∈R, then v2∈B, v3∈R …vn∈B (note that n is even number). So there is no violation (R-B-R…-B-R). You can cycle with no problem.

2). the length of C is *odd*. Then C = {v1-v2-…-vn-v1} (n is odd) let v1∈R, then v2∈B, v3∈R …vn∈R (note that n is even number). So there is a violation (R-B-R…R-R). It contradicts that G is a bipartite graph.

(3**). BFS for detecting the odd cycle**: Since then, the existence of odd cycle is easily found with BFS algorithm. First, enqueue a certain source node in queue and color it with one color, say Red. This node is considered as already visited. Then, all of adjacent nodes are enqueued while being colored with different color, say Blue and marked as visited. This process is repeated until there is no other unvisited nodes, that is queue is empty. While checking all of adjacent nodes to dequeued node v, if you visit already marked node and its color is same with that node v, it means an odd cycle: Or you can say this odd cycle from the viewpoint of level. Given that the level of source node is considered as 0 and level of adjacent nodes, say its children is incremented by 1, there exists an odd cycle if there is a certain edge whose ends are connected to two nodes that lies in same level. Note that the number of source node is one so such an edge guarantees the existence of an odd cycle in that graph: whether you detect the odd cycle as same color between two adjacent color or the existence of an edge in same level, you can find out successfully its bipartiteness. In case of unconnected graph G, however, note that all of components need to be handled. If there exists no odd cycle in these graph, this graph is a bipartite graph.

Below is a more detailed description:

1). *Color source node v with R*: a certain node can be a source node. Let this be node v. first it is considered as visited and enqueued in queue. Its color is also considered as a certain color, say Red.

2). *Visit adjacent nodes and color*: the source node is dequeued from the front of queue and its adjacent nodes are enqueued in queue, being colored with different color of that node v (that is Blue) and marked as visited and so on. Until all of nodes in this graph are visited, (that is queue becomes empty) this process is iterated: if you visit a certain node which is already marked and its color is same with just dequeued node, it means that an odd cycle occurs.

In conclusion, the answer for this decision problem is considered as No and just return it. If there is no odd cycle until queue becomes empty, which means this graph G is a bipartite graph and then finally Yes is returned. Beware that all of components should be checked if G is not a connected graph.

This is how 2-coloring problem works using BFS.

Ⅲ. Algorithm Summary for 34.1 Polynomial time

Three are three properties to formalize notion of polynomial time solvable problems: First, time complexity would be improved once one algorithm for a certain problem is discovered. Second, there are various solution models for one problem. Third, the polynomial time is closed under sum, product, and composition operations.

(1). **Abstract problem**:An abstract problem Q is composed of two problem sets: a set of *instances I* and a set of *solutions S*. with this, decision problems would be considered as a function which assign I to S {1,0} (=Yes/No). In case of shortest path problem, this function tells whether a certain path from node u to node v exists or not: even though this feature is not enough to compute a specific max or min value, which this optimization problem requires, these could be also redefined as decision problems.

(2). **Encodings**: An *encoding* of a set S assigns e in S to the set of binary strings. All of compound objects including polygon, function and program can be encoded: So a computer algorithm encodes a problem instance I as input. In this case, if a certain problem’s instance set is a set of binary strings, it said as a *concrete problem*, which is solvable in time O(T(n)). If it is solvable in time *O(nk)* with a certain algorithm, this concrete problem is polynomial-time solvable.

A function f : {0,1}\* → {0,1}\* (denoted as the set of all strings composed of symbols from the set {0,1}) is polynomial-time computable if it produces from input x ∈ {0,1}\* output f(x). and two encodings e1 and e2 are polynomially related, given that f12 and f21 such that f12(e1(i))=e2(i) and f21(e2(i))=e1(i) for an element i of set I: if there exists such polynomially related e1 and e2, the definition of polynomial-time solvability is independent of any particular encoding types: that is, if then, e1(Q)∈P ↔ e2(Q)∈P.

(3). **A formal-language framework**: These is the machinery of formal-language theory: 1). *∑ (alphabet)*, a finite set of symbols 2). *L (language)* over ∑, a set of strings made up of symbols from ∑ 3). *(the empty string)* 4). *∑\**, the language of all strings over alphabet. In addition to these definitions, there are various operations such as union and intersection and set-theoretic definitions such as complement and concatenation. Plus, the closure (or Kleene star) of a language *L* refers to *L*\* =

By this formal-language framework, the relationship between decision problems and their solution algorithms is explained: 1). an algorithm *A* *accepts* a string x∈{0,1}\*, if *A*(x) is 1 with input x. otherwise, 2). *A* *rejects* a string x if A(x) is 0. Then, 3). the language *accepted* by *A* is the set of strings *L* = {x∈{0,1}\* : *A*(x)=1}. Additionally, 4). a language *L* is *decided* by *A* if every binary string is accepted by *A* and not in *L* is rejected by *A*. 5). *L* is *accepted in polynomial time* by *A* if it is accepted by *A* and a certain constant *k* exists, where *A* accepts length-n string x in time O(nk). 6). *L* is *decided in polynomial time* by *A* if *k* exists, where *A* decides whether length- n string x(x belongs to {0,1}\*)∈*L* in time O(nk). 7). A *complexity class P* = {*L*⊆{0,1}\* : an algorithm *A* that decides *L* in polynomial time exists} = {*L* : *L* is accepted in polynomial time} In conclusion, the existence of such a bound on the time consuming (running time) for *L*∈*P* would guarantee the existence of a certain algorithm which checks that bound, whether it can be discovered or not.

Ⅳ. Explanation for algorithms

Ⅰ. **IDE**: Microsoft Visual Studio Professional 2013

Ⅱ. **Explanations for algorithms**: A user-driven program for 5 algorithms. There are two types of graph: 1). Directed graph 2). Undirected graph. Since a certain type of graph is to be required in a certain graph algorithm, it is checked before invoking these algorithms: directed graph for Warshall’s, Floyd’s and Dijkstra’s algorithm and undirected graph for Dijkstra’s, Kruskal’s and Prim’s algorithm. In case of Dijkstra’s algorithm, both of them can be used so that you could select which graph is implemented.

\* Implementation of graph: graph is basically implemented in adjacent linked list. With your entered number of node(=vertex)s and edges, each header referring to a certain node is generated(i.e. header node means ‘from’ node) and adjacent node (i.e. ‘to’ node) is added to refer to the existence of an edge between them. In case of undirected graph, they are connected in both directions. A certain weight(=cost) of this edge is also added.

e.g. (1, 2, 3: (an edge from node1 to node2 of weight 3 in directed graph/an edge from node1 to node2 and from node2 to node1 of weight 3 in undirected graph)

\*notification for the number of node numNode, numEdge and weight

0<numNode<MAX\_SIZE

0<=numEdge<=(numNode)\*(numNode-1) in case of directed graph

0<=numEdge<=(numNode)\*(numNode-1)/2 in case of undirected graph

0<weight<=MAX\_SIZE

Self-loop and parallel edge would be excluded. MAX\_SIZE is set as 1000 in this C code.

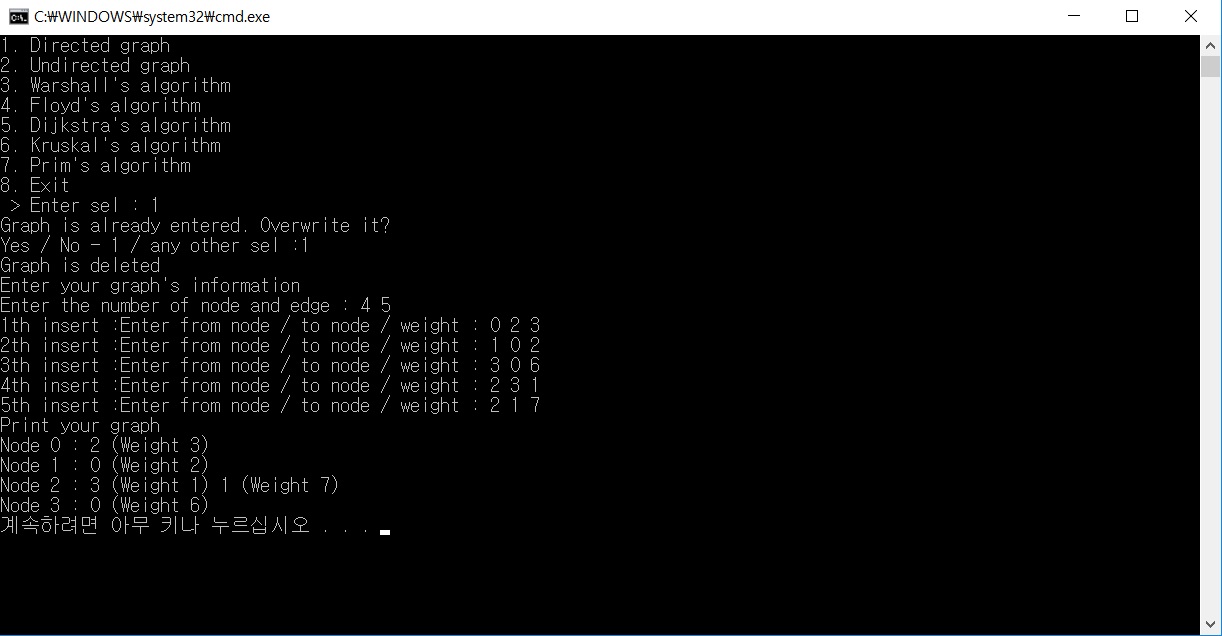
In order to illustrate more explicitly, adjacent matrix is generated and then printed out in accordance with a certain algorithm such as Warshall’s and Floyd’s algorithm.

\*For more detailed code, please see C code and exe file in submitted zipped file.

Input data of directed and undirected graphs are below:

1). Directed graph

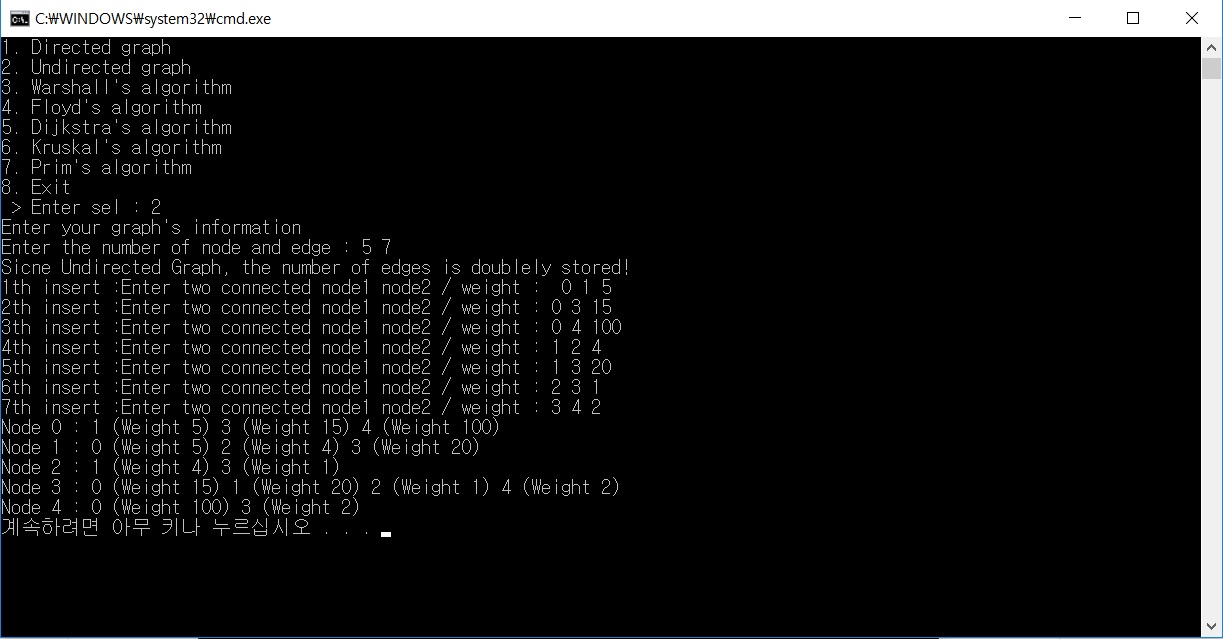
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Node0 | Node1 | Node2 | Node4 |
| Node0 | 0 | NIL | 3 | NIL |
| Node1 | 2 | 0 | 0 | NIL |
| Node2 | NIL | 7 | 0 | 1 |
| Node3 | 6 | NIL | NIL | 0 |



\*1). Adjacent list of input data of directed graph.

2). Undirected graph

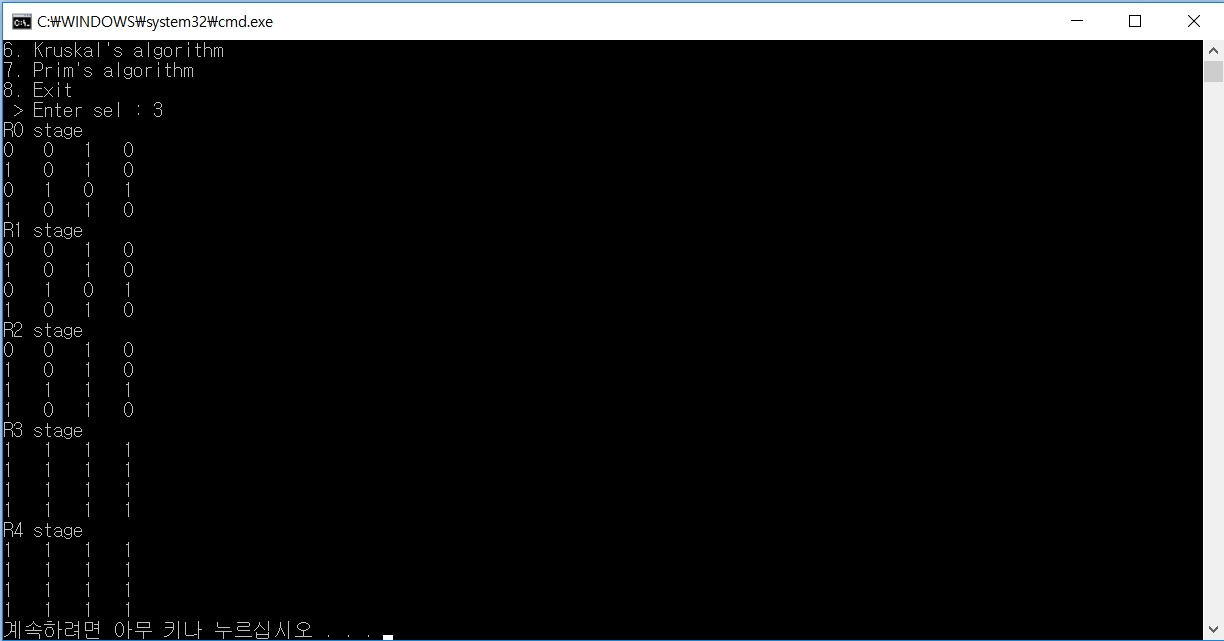
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Node0 | Node1 | Node2 | Node3 | Node4 | Node5 |
| Node0 | 0 | 1 | NIL | 7 | NIL | NIL |
| Node1 | 1 | 0 | NIL | 4 | 2 | NIL |
| Node2 | NIL | NIL | 0 | 8 | NIL | NIL |
| Node3 | 7 | 4 | 8 | 0 | 5 | 6 |
| Node4 | NIL | 2 | NIL | 5 | 0 | 3 |
| Node5 | NIL | NIL | NIL | 6 | 3 | 0 |



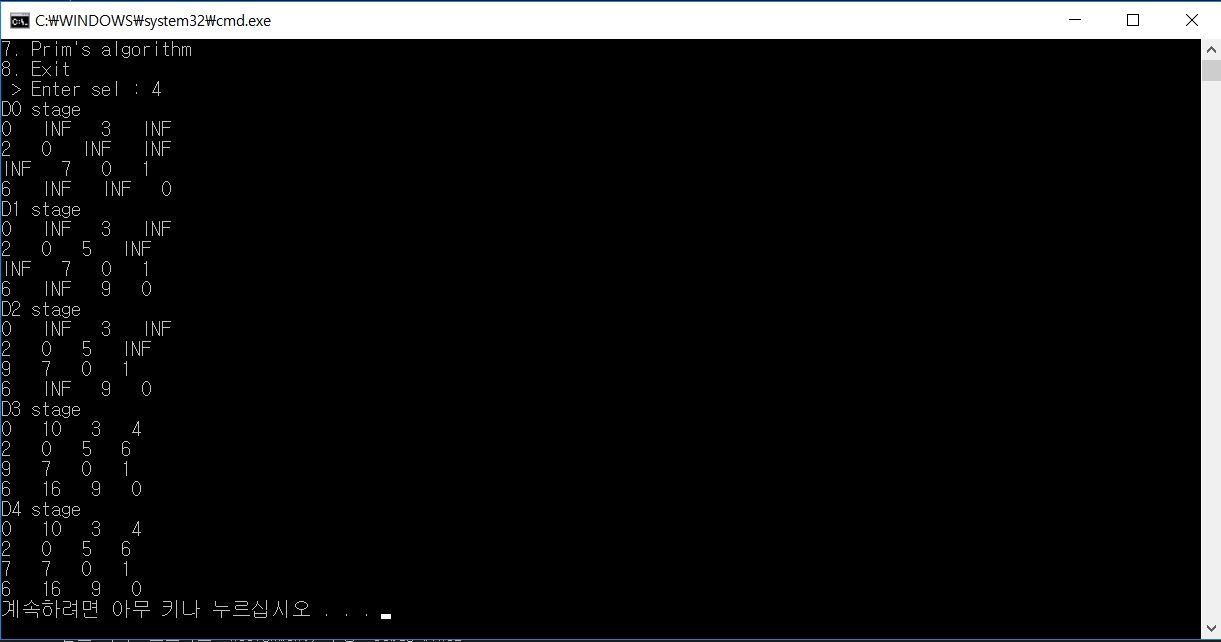
\*2). Adjacent list of input data of undirected graph.

(1). **Warshall’s algorithm**: With entered graph, a certain adjacent matrix is generated in accordance with it. If there is an edge between two nodes, check it as 1, since all of weights is assumed to be positive in this program. Then with triple nested for loops, the transitive closure is computed from R0 to to Rn steps. Here n indicates the number of node in directed graph. The key operation is: mat[i][j]←MAX(mat[i][k], mat[i][k] && mat[k][j])

\*Screenshot for implementation of Warshalls’ algorithm

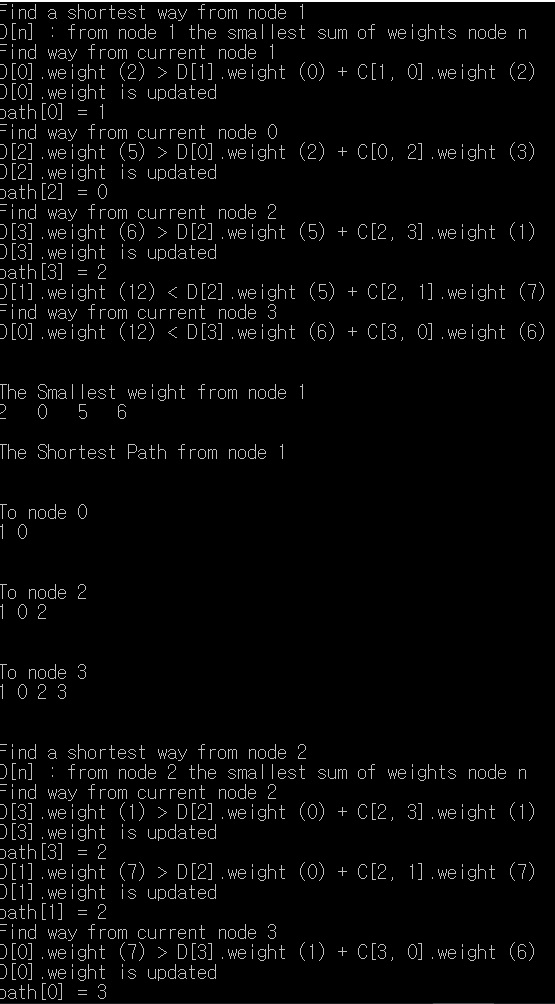
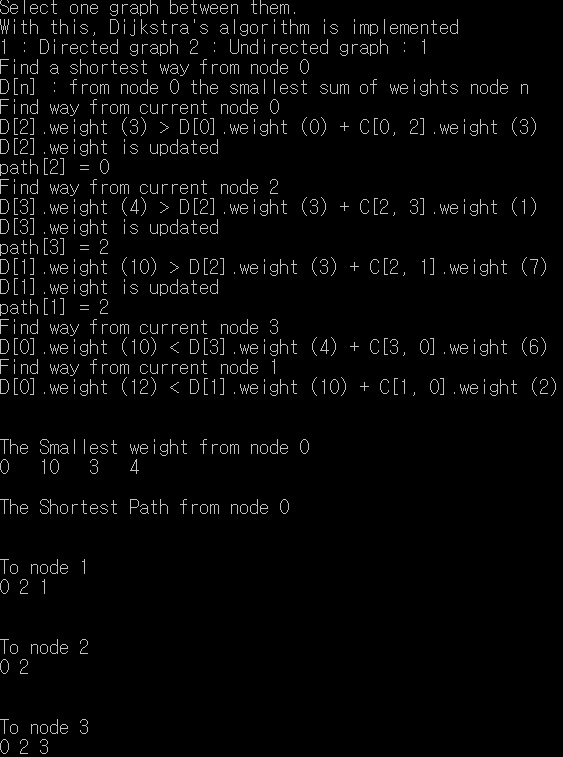


(2). **Floyd’s algorithm**: With newly generated adjacent matrix as same as in Warshall’s algorithm, all pairs of shortest distances are computed using triple nested for loops. The key operation is: mat[i][j]←MIN(mat[i][k], mat[i][k] + mat[k][j]). If there is no possible path to a certain node in a certain step k, its weight is set as Infinity(here set as MAX\_SIZE) and updated as well. \*Screenshot for implementation of Floyd’s algorithm.

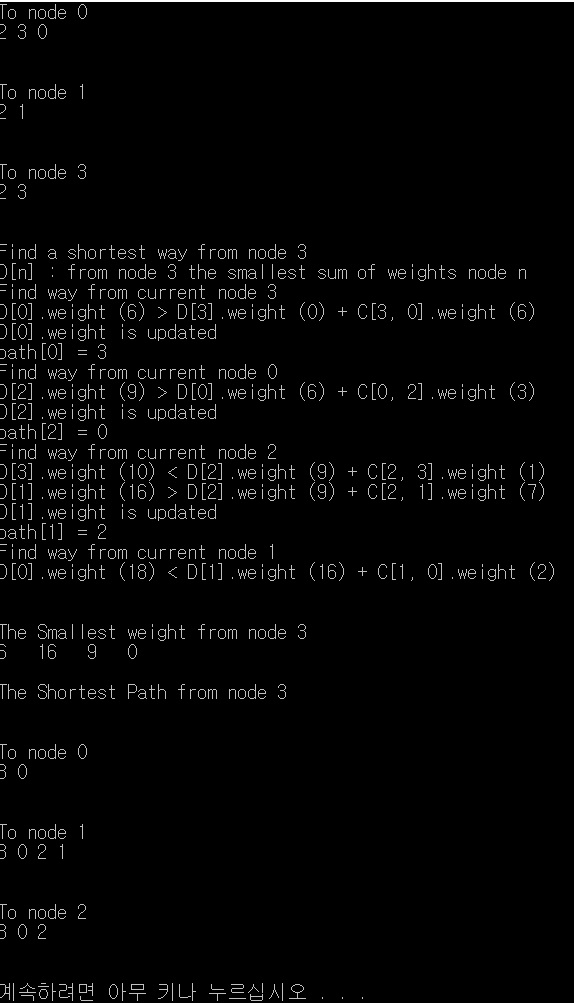
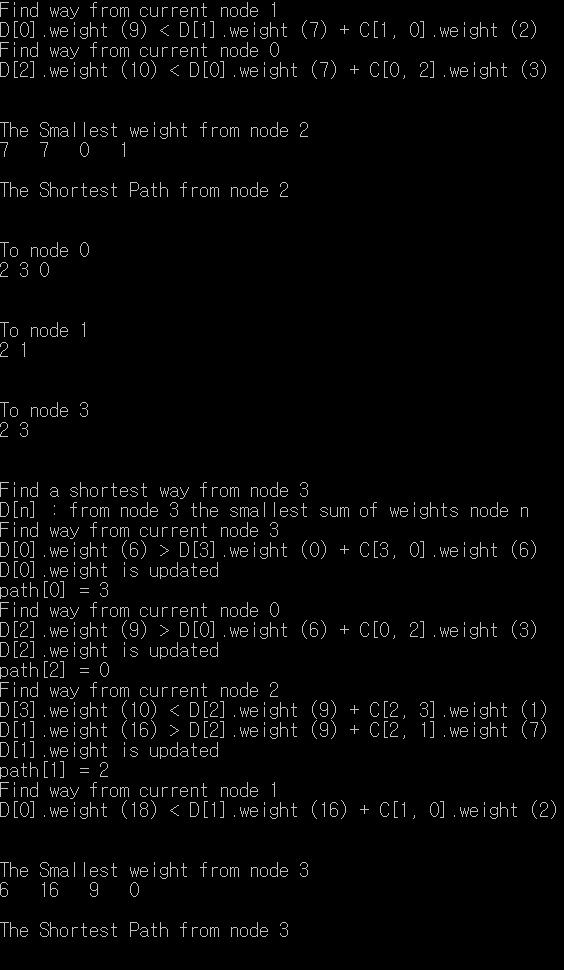


(3). **Dijkstra’s algorithm**: with either of two graphs, the shortest distances from a single source is computed. With min-priority queue, the possible shortest distances from a single node are found. In this implementation, the shortest paths from all of nodes (node 0~node i) are computed. At first moment, an array D[k] is set as MAX\_SIZE except for D[i] at stage of node i and node i is enqueued with weight 0 in priority queue as well. From them, all of possible path is compared and updated until priority queue is empty, backtracking its path too. After all of shortest paths are recorded in D[k], its distances and each paths are printed out.

\*Screenshot for implementation of Dijkstra’s algorithm. In this case you will see only a case of directed graph, which is handled above.

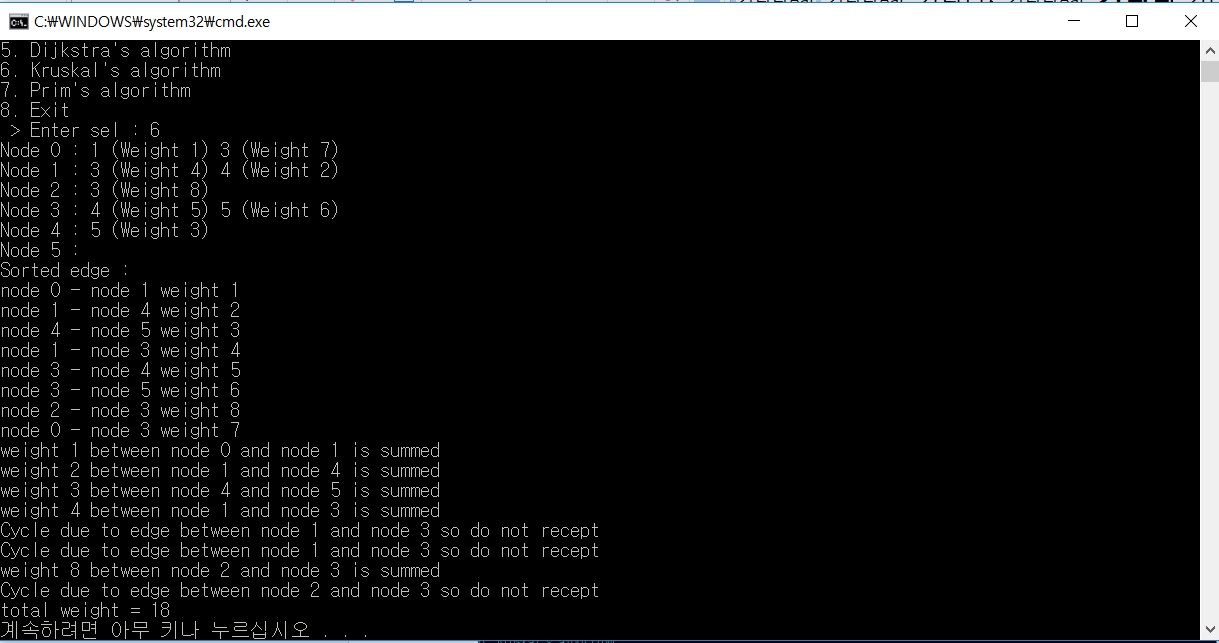


From node 0 to node 3 (there is total 4 nodes in this directed graph), all of shortest path and weights is computed.



(4). **Kruskal’s algorithm**: With sorted edges in increasing order of entered undirected (connected) graph using min priority queue, a certain weight of edge is summed using union and find set operations . Union operation is implemented using an array, each indicating that a certain node is related with another node. By checking these with Find operation, duplicate edge would be prevented in advance.

\*Screenshot for implementation of Kruskal’s algorithm. Make note of an undirected graph above.



(5). **Prim’s algorithm**: From a certain source node you entered in advance, every node adjacent to it are checked in order of less weight using min priority queue. Until priority queue is empty, the smallest weight adjacent to that node is summed and its adjacent nodes are enqueued, while visited node is skipped.

\*Screenshot for implementation of Prims’ algorithm. Even if total sum is same as in case of Kruskal’s algorithm, the remaining graph could be different depending on starting source node.

