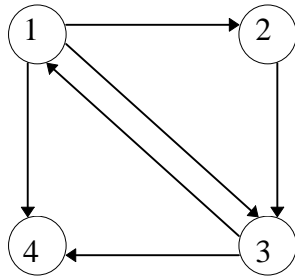


3. Basic Graph Algorithms

3.1 Introduction

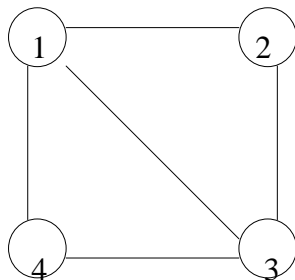
A graph G is defined by $G = (V, E)$ where V is the set of vertices and E is the set of edges. The set E is a subset of the direct product $V \times V$. Conventionally we express as $|V| = n$ and $|E| = m$. For convenience of computer implementation, we express vertices by integers, that is, $V = \{1, \dots, n\}$. This definition of graph is mathematically complete, but not very good for human consumption. If we express vertices by points and edges by arrows, we can visualise the given graph.

Example 1. $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (3, 1), (3, 4)\}$.



In the above definition we distinguish between (u, v) and (v, u) , that is, we define directed graphs. If we ignore direction over edges, we have an undirected graph, in which we regard (u, v) and (v, u) as the same thing, denoted by $[u, v]$.

Example 2. If we ignore direction in Example 1, we have $G = (V, E)$, where $V = \{1, 2, 3, 4\}$ and $E = \{[1, 2], [1, 3], [1, 4], [2, 3], [3, 4]\}$.



A path is a sequence of edges $(v_0, v_1)(v_1, v_2) \dots (v_{k-1}, v_k)$, which is called a path from v to v of length k . For convenience we define the null for the case of $k = 0$. There is the null path from v to v for any vertex v . For simplicity the above path is expressed as (v_0, v_1, \dots, v_k) . If all v_i 's are distinct in the path, it is called a simple path. If $v_0 = v_k$, for $k > 0$, this path is called a cycle. If all vertices are distinct except for the first and the last

in a cycle, this cycle is called a simple cycle. In an undirected graph similar concepts are defined using $[u, v]$ in stead of (u, v) .

Example 3. In Example 1, $(1, 2, 3, 4)$ is a simple path, $(1, 2, 3, 1)$ is a simple path, $(1, 2, 3, 1, 4)$ is a non-simple, path, and $(1, 2, 3, 1, 3, 1)$ is a non-simple cycle.

Example 4. For graph in Example 2, $(1, 2, 3, 4)$ is a simple path and $(1, 2, 3, 4, 1)$ is a simple cycle.

Now we define reachability. If there is a path from u to v we say v is reachable from u and write as $u \rightarrow v$. Since there is a null path from v to v for any v , we have $v \rightarrow v$. If $u \rightarrow v$ and $v \rightarrow u$, we say u and v are mutually reachable and write as $u \leftrightarrow v$. This relation \rightarrow is an equivalence relation on the set V . The equivalence classes defined by \rightarrow are said to be strongly connected (sc) components of the graph. The vertices in the same equivalence class are mutually reachable. In an undirected graph, there is no distinction between \rightarrow and \rightarrow . Equivalence classes are called connected components.

Example 5 For the graph in Example 1, we have sc-components $\{1, 2, 3\}$ and $\{4\}$. For the graph in Example 2, V itself is a single connected component.

Now we investigate into data structures suitable for graphs to be processed by a computer.

Let

$$\begin{aligned} \text{OUT}(v) &= \{ w \mid (v, w) \text{ is in } E \} \\ \text{IN}(v) &= \{ u \mid (u, v) \text{ is in } E \}. \end{aligned}$$

$|\text{OUT}(v)|$ ($|\text{IN}(v)|$) is called the out (in) degree of v . $\text{OUT}(v)$ (or $\text{IN}(v)$) is called the adjacency list of v . The adjacency list expression of the given graph takes $O(m+n)$ space. We can exhaust edges from each vertex in $O(1)$ time per edge. It takes $O(n)$ time in the worst case to check whether there is an edge from u to v .

The next data structure is an adjacency matrix $A = [a_{ij}]$, where

$$\begin{aligned} a_{ij} &= 1, \text{ if there is an edge } (i, j) \\ &0, \text{ otherwise.} \end{aligned}$$

In this expression, we can check if (i, j) is an edge in $O(1)$ time, although we need $O(n^2)$ space. If the graph is sparse, that is, $m \ll n^2$, this expression by adjacency lists is expensive.

Example 6. For the graphs in Examples 1 and 2, we have

$$A = \begin{matrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{matrix}$$

$$\begin{array}{r}
 A = \begin{array}{cccc}
 0 & 1 & 1 & 1 \\
 1 & 0 & 1 & 0 \\
 1 & 1 & 0 & 1 \\
 1 & 0 & 1 & 0
 \end{array}
 \end{array}$$

3.2 Depth-first Search and Breadth-first Search

We consider the problem of visiting all vertices in a graph starting from some designated vertex. There are two representative search methods; depth-first (DFS) and breadth-first (BFS). In the following we give the visit number to each vertex.

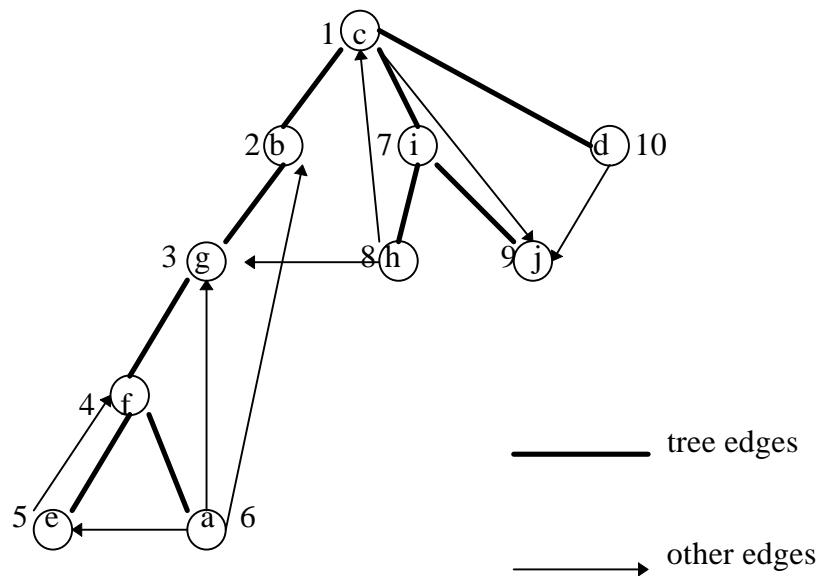
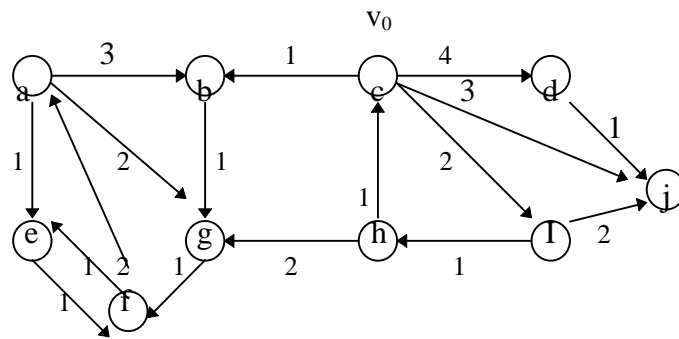
Depth-first search. We start from a vertex and go straight forward to the dead end. Then come back to the previous vertex and again go straight and so forth. We keep track of visited vertices and traversed and examined edges so that we will not examine those things again. If we come back to the starting vertex and have exhausted all edges, we finish the search from the starting vertex. If there are still unvisited vertices, we hop to an unvisited vertex and repeat the same procedure. The algorithm follows.

1. **procedure** DFS(v);
2. **begin** mark v “visited”;
3. $N[v] := c$; $c := c + 1$;
4. **for** each w in $OUT(v)$ **do**
5. **if** w is “unvisited” **then begin**
6. Add edge (v, w) to T ;
7. DFS(w)
8. **end**
9. **end**;
10. **begin** {main program}
11. $T := \emptyset$; { T is the set of edges actually traversed}
12. $c := 1$; { c is the counter for visit numbers}
13. **for** v in V **do** mark v “unvisited”;
14. **while** there is an unvisited v in V **do** DFS(v)
15. **end**.

$N[v]$ is the visit number of vertex v . If there is an edge to an unvisited vertex w , we go to w by the recursive call line 7. If we exhaust all edges or $OUT(v)$ is empty from the beginning, we finish DFS(v) and go back to the previous vertex. If we finish one DFS at line 14, the traversed edges which are recorded in T form a tree. If we do DFS by changing v at line 14, we form several trees, that is, a forest, which is called a depth-first spanning forest of the given graph G .

Example 1. Apply the algorithm to the following graph starting from v_0 . The numbers on edges show the order in which vertices are placed in $OUT(v)$ for each v . Note that edges are examined in that order. The edges in T are shown by solid arrows. Dotted

arrows are non-traversed edges, that is, only examined. The numbers beside the circles show the visit numbers. In this example we have only one tree in the forest.



After depth-first search, edges are classified into four categories.

- (1) Tree edges; edges recorded in T which are actually traversed.
- (2) Forward edges; non-traversed edges which go from ancestors to descendants.
- (3) Back edges; non-traversed edges which go from descendants to ancestors.
- (4) Cross edges; non-traversed edges which go between vertices that are neither ancestors nor descendants of one another.

Note that a cross edge goes from a vertex whose visit number is greater to a vertex whose visit number is smaller.

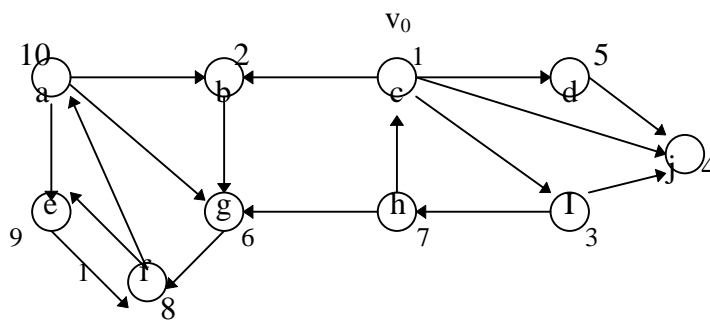
Example (c, j) is a forward edge. (h, g), (d, j) and (a, e) are cross edges. (e, f), (a, g), (a, b) and (h, c) are back edges.

Breadth-first search. In depth-first search, we use a stack implicitly by the recursive call to keep track of the search history. If we use a queue, or a first-in-first-out data structure, we will have the following breadth-first algorithm.

1. procedure BFS(v);
2. begin
3. $Q := \{v\}$; /* Q is a queue */
4. while $Q \neq \emptyset$ do begin
5. $v \leftarrow Q$; /* take out from Q */
6. Mark v “visited”;
7. $N[v] := c$; $c := c + 1$;
8. for each w in OUT(v) do
9. if w is “unvisited” and w is not in Q then
10. $Q \leftarrow w$ /* push into Q */
11. end
12. end;
13. begin {main program}
14. for v in V do mark v “unvisited”;
15. $c := 1$;
16. while there is an unvisited v in V do BFS(v)
17. end.

As a by-product we can compute shortest distances to vertices reachable from the starting vertex.. If we record traversed edges, we can compute shortest paths as well.

Example 2. If we apply BFS to the graph in Example 1, we have the following visit numbers. After we process OUT(b), the status of Q is given by $Q = (i, j, d, g)$.



3.3 Strongly Connected Components

The concept of depth-first search is applied to find strongly connected components in this section. Let the sets of strongly connected components be V_i ($i=1, \dots, k$). Let the set of edges E_i be defined by $E_i = \{ (u, v) \mid u \text{ and } v \text{ are in } V_i \}$. Then we have the following lemma.

Lemma. Graph $(V_i, E_i \cap T)$ form a tree.

Example. In Example 1, we have $V = \{b, g, f, e, a\}$, $V = \{j\}$, $V = \{d\}$, $V = \{c, i, h\}$. Observe that each sc-component forms a tree in the depth-first spanning forest.

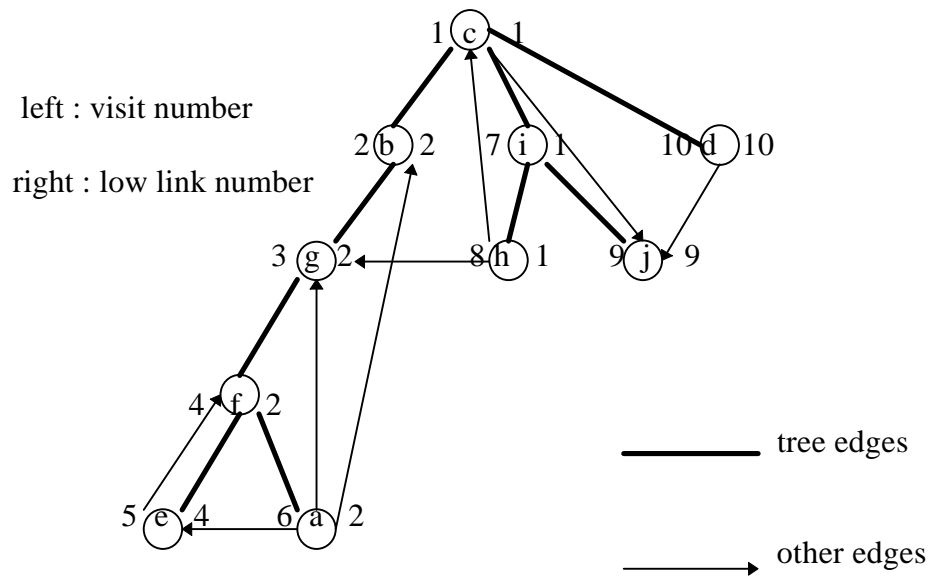
If we can know the root of each sc-component, we can harvest it when we finish DFS from that vertex. If we can reach from v to vertex with a lower visit number, v can not be the root of an sc-component. We record this reachability in array L . Also accumulated candidates for sc-components are recorded in $STACK$, a stack data structure, that is, first-in-last-out data structure. The algorithm follows.

1. **procedure** CONNECT(v);
2. **begin** Mark v “visited”;
3. $N[v] := c$; $c := c + 1$;
4. $L[v] := N[v]$; $STACK \leftarrow \{v\}$; /* push v to $STACK$ */
5. **for** each w in $OUT(v)$ **do**
6. **if** w is “unvisited” **then begin**
7. CONNECT(w);
8. $L[v] := \min\{L[v], L[w]\}$;
9. **end**
10. **else if** $N[w] < N[v]$ **and** w is in $STACK$ **then**
11. $L[v] := \min\{N[w], L[v]\}$;
12. **if** $L[v] = N[v]$ **then begin**
13. Take vertices out of $STACK$ until v and write them
14. **end**;
15. **begin** {main program}
16. $c := 1$; $STACK := \emptyset$;
17. **for** each v in V **do** mark v “unvisited”;
20. **while** there is an “unvisited” vertex v **do** CONNECT(v)
21. **end**.

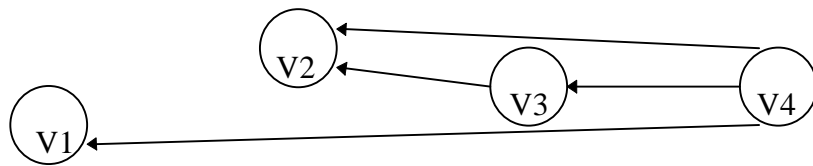
Example. If we apply CONNECT to the graph in Example 1, we have

$$V_1 = \{a, e, f, g, b\}, V_2 = \{j\}, V_3 = \{d\}, V_4 = \{h, i, c\}$$

in this order. In the following figure, visit numbers are attached to the right of circles and those of L are attached to the right. L is called the low link.



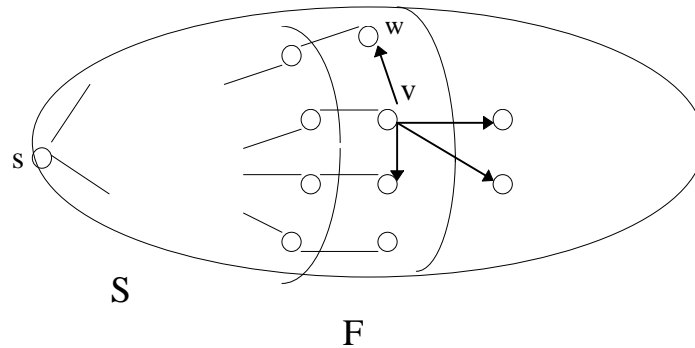
If we crush each sc-component into a single vertex and edges connecting them into single edges, we have an acyclic graph as shown below.



4 Shortest Path Algorithms

4.1 Single Source Problem -- Dijkstra's Algorithm

We consider the problem of computing the shortest paths from a designate vertex s , called the source, to all other vertices in the given graph $G = (V, E)$. For simplicity we assume all other vertices are reachable from s . We maintain three kinds of sets, S , F , and $V-S-F$, called the solution set, the frontier set, and the unknown world. See below.



The set S is the set of vertices to which the algorithm computed the shortest distances. The set F is the set of vertices which are directly connected from vertices in S . The set $V-S-F$ is the set of all other vertices. We maintain the distance to each vertex v , $d[v]$, which is the shortest distance to v if v is in S . The distance $d[v]$ for v in F is defined as follows:

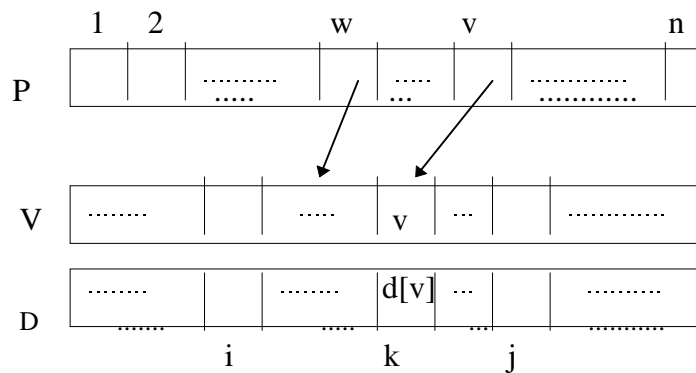
$d[v]$ = distance of the shortest path that lies in S except for the end point v .

If we find the minimum such $d[v]$ for v in F , there is no other route to v which is shorter. Thus we can include v in S . After this we have to update the distances to other vertices in F , and set up distances to vertices in $V-S-F$ that are directly connected from v and include them in F . Let $OUT(v) = \{ w \mid (v, w) \text{ is in } E \}$.

1. $S := \emptyset$;
2. Put s into S ; $d[s] := 0$;
3. **for** v in $OUT(s)$ **do** $d[v] := L(s, v)$; $\{ L(u, v) \text{ is the length of edge } (u, v) \}$
4. $F := \{ v \mid (s, v) \text{ is in } E \}$;
5. **while** F is not empty **do begin**
6. $v := u$ such that $d[u]$ is minimum among u in F
7. $F := F - \{v\}$; $S := S \cup \{v\}$;
8. **for** w in $OUT(v)$ **do**
9. **if** w is not in S **then**
10. **if** w is in F **then** $d[w] := \min \{ d[w], d[v] + L(v, w) \}$
11. **else begin** $d[w] := d[v] + L(v, w)$; $F := F \cup \{w\}$ **end**
12. **end.**

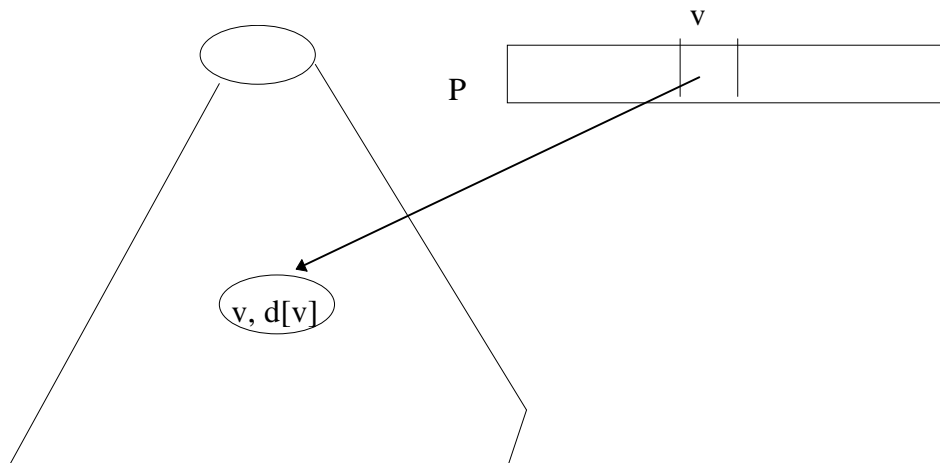
Let the set V of vertices be given by the set of integer $\{1, \dots, n\}$. The simplest way to implement the data structures is that we maintain the distances in array d and the set S in Boolean array S ; $S[v] = \text{true}$ if v is in S , $S[v] = \text{false}$ otherwise. The distances to v in $V - S - F$ is set as $d[v] = \infty$. The set F can be identified by element v such that $S[v] = \text{false}$ and $d[v] = \infty$. Line 6 can be implemented by scanning array d and testing the Boolean condition S , taking $O(n)$ time. Lines 8-11 can be implemented by scanning the list $\text{OUT}(v)$, taking $O(n)$ time in the worst case. The overall time will be $O(n^2)$.

A slightly better implementation is described next. The solution set S is maintained in array V with indices $1 \dots I$. The set F is with indices $I+1 \dots j$. The contents of V are actual vertices. The array P is for pointing to the positions of vertices in V . See below.



We find the minimum $d[v]$ at line 6, which is found at k , and swap $(V[i], D[i])$ with $(V[k], D[k])$ and increase I by 1. The corresponding element of P is updated. We also add new elements to V and D for inclusion in F and increase j accordingly. The pointers to the new elements in F will be set up in P . This implementation is efficient when the size of F is not large as in planar graphs such as maps. Also this implementation is a good introduction to the more sophisticated implementation with priority queues.

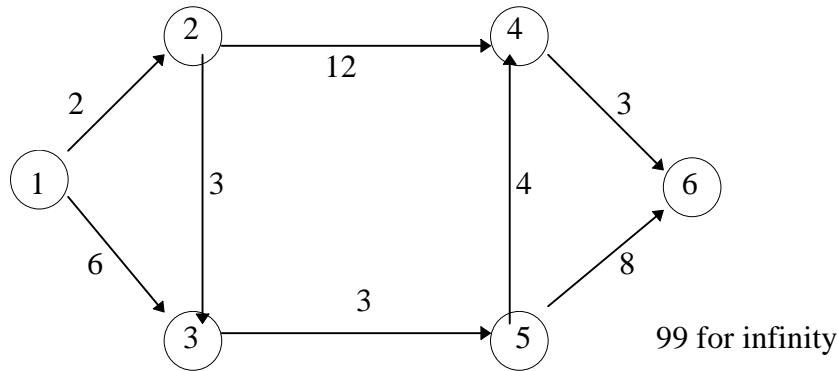
An implementation with a binary heap is described next. The idea is similar to the above. The pairs $(v, d[v])$ are maintained in a binary heap with $d[v]$ as a key. See below.



The n minimum operations at line 6 will take $O(n \log n)$ time. Suppose there are m edges. Then the total decrease_key operations and insert operations at lines 7-11 will take $O(m \log n)$ time. Thus the total running time is $O(m \log n)$, since if we can reach all vertices from s , we can assume $n-1 \leq m$. This implementation is efficient when the graph is sparse such as planar graphs where $m = O(n)$. The time becomes $O(n \log n)$ in this case.

If we use the heap as a priority queue, we can solve the problem in $O(m+n \log n)$ time.

Example Let us trace the changes of array d for the following graph.



0	2	6	99	99	99
1	2	3	4	5	6

$v=2, S=\{1\}, F=\{2, 3\}$

0	2	5	14	99	99
---	---	---	----	----	----

$v=3, S=\{1, 2\}, F=\{3, 4\}$

0	2	5	14	8	99
---	---	---	----	---	----

$v=5, S=\{1, 2, 3\}, F=\{4, 5\}$

0	2	5	12	8	16
---	---	---	----	---	----

$v=4, S=\{1, 2, 3, 5\}, F=\{4, 6\}$

0	2	5	12	8	15
---	---	---	----	---	----

$v=6, S=\{1, 2, 3, 5, 4\}, F=\{6\}$

0	2	5	12	8	15
---	---	---	----	---	----

4.2 Transitive Closure and the All Pairs Shortest Path Problem

Let A be the adjacency matrix of the given graph G . That is

$$A[i, j] = 1, \text{ if } (i, j) \text{ is in } E \\ 0, \text{ otherwise.}$$

The reflexive-transitive closure of A , A^* , is defined as follows:

$$A^*[i, j] = 1, \text{ if } j \text{ is reachable from } i \\ 0, \text{ otherwise.}$$

The transitive closure of A , A^+ , is defined as follows:

$$A^+[i, j] = 1, \text{ if } j \text{ is reachable from } i \text{ over a path of at least one edges} \\ 0, \text{ otherwise.}$$

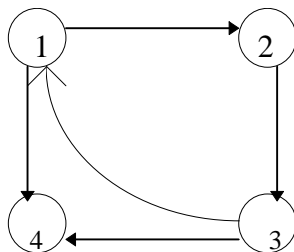
The computations of both closures are similar. In the following we describe the transitive closure. The algorithm was invented by Warshall in 1962.

1. **for** $i:=1$ **to** n **do** **for** $j:=1$ **to** n **do** $C[i, j] := A[i, j]$;
2. **for** $i:=1$ **to** n **do** $C[i, i] := C[i, i] + 1$
3. **for** $k:=1$ **to** n **do**
4. **for** $i:=1$ **to** n **do** **for** $j:=1$ **to** n **do**
5. $C[i, j] := C[i, j] \vee (C[i, k] \wedge C[k, j])$

Theorem. At the end of the algorithm, array C gives the transitive closure A^* .

Proof. Theorem follows from mathematical induction on k . We prove that $C[i, j] = 1$ at the end of the k -th iteration if and only if j is reachable from i only going through some of the via set $\{1, 2, \dots, k\}$ except for endpoints i and j . The basis of $k=0$ is clear since the via set is empty. Now the theorem is true for $k-1$. We observe that we can go from i to j via $\{1, 2, \dots, k\}$ if and only if we can go from i to k via $\{1, \dots, k-1\}$ or we can go from i to k via $\{1, \dots, k-1\}$ and from k to j via $\{1, \dots, k-1\}$. Thus line 5 correctly computes C for k .

Example. We trace array C for each k for the following graph.



	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
$C =$	1 1 0 1	1 1 0 1	1 1 1 1	1 1 1 1	1 1 1 1
	0 1 1 0	0 1 1 0	0 1 1 0	1 1 1 1	1 1 1 1
	1 0 1 1	1 1 1 1	1 1 1 1	1 1 1 1	1 1 1 1
	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1

For hand calculation, it is convenient to enclose the k -th row and column in the $(k-1)$ th array to get the k -th array. To get the (i, j) element in the k -th array, we go horizontally and vertically to the enclosed axes. If we find 1's in both axes, we can set up 1 if the original element was 0. See below.

$k = 0$

1	1	0	1
0	1	1	0
1	0	1	1
0	0	1	1

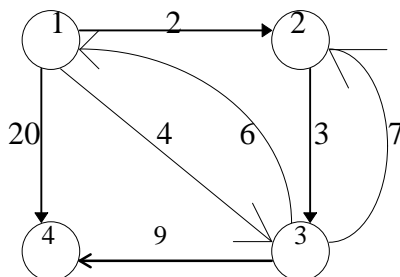
Now we turn to the all pairs shortest path problem. Let $D[i, j]$ be the given distance from vertex i to vertex j . Let $D[i, j]$ be ∞ if there is no edge from i to j . The following algorithm for all pairs problem was invented by Floyd in 1962.

1. for $i:=1$ to n do for $j:=1$ to n do $C[i, j] := D[i, j]$;
2. for $i:=1$ to n do $C[i, i] := 0$;
3. for $k:=1$ to n do
4. for $i:=1$ to n do for $j:=1$ to n do
5. $C[i, j] := \min\{C[i, j], C[i, k] + C[k, j]\}$

Theorem. At the end of the algorithm, array element $C[i, j]$ gives the shortest distance from i to j .

Proof. Similar to that of Theorem 1. We can prove by induction that $C[i, j]$ at the end of the k -th iteration gives the distance of the shortest path from i to j that lies in the via set $\{1, \dots, k\}$ except for endpoints i and j . The details are left with students.

Example. We trace the matrix C at each iteration by k for the following graph.



	k=0	k=1	k=2	k=3	k=4
C =	0 2 6 20 ∞ 0 3 ∞ 4 7 0 9 ∞ ∞ ∞ 0	0 2 6 20 ∞ 0 3 ∞ 4 6 0 9 ∞ ∞ ∞ 0	0 2 5 20 ∞ 0 3 ∞ 4 6 0 9 ∞ ∞ ∞ 0	0 2 5 14 7 0 3 12 4 6 0 9 ∞ ∞ ∞ 0	0 2 5 14 7 0 3 12 4 6 0 9 ∞ ∞ ∞ 0

The axes are enclosed here also to ease calculations. Specifically $C[3, 2]$ at $k=1$ is obtained by $6 = \min\{7, 4+2\}$ in C at $k=0$.

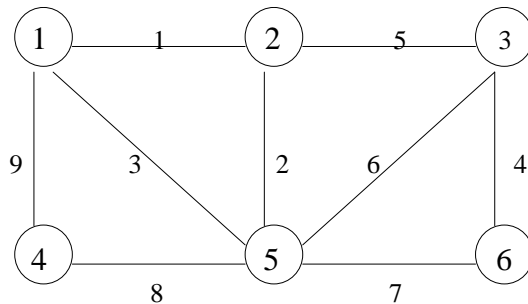
As we see from the above two algorithms, they are very similar. In fact, these two problems can be formulated by a broader perspective of semiring theory. That is, Boolean algebra and the distance calculation both satisfy the axioms of semiring. The general algorithm will calculate the closure of a given matrix over the semiring. See for more details the book “Design and Analysis of Computer Algorithms” by Aho, Hopcroft and Ullman published in 1974.

5 Minimum Cost Spanning Trees

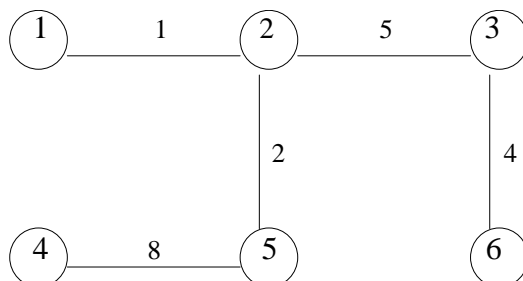
5.1 Introduction

The minimum cost spanning tree (abbreviated as minimum spanning tree) of a given undirected graph G is a spanning tree of G with the minimum cost. A spanning tree is a tree that connect all the vertices. The cost of a spanning tree is the sum of the edge cost. An application of this problem is seen in the city planning that will locate a public facility that satisfy all the needs of the city residents and minimise the cost of the road construction.

Example.

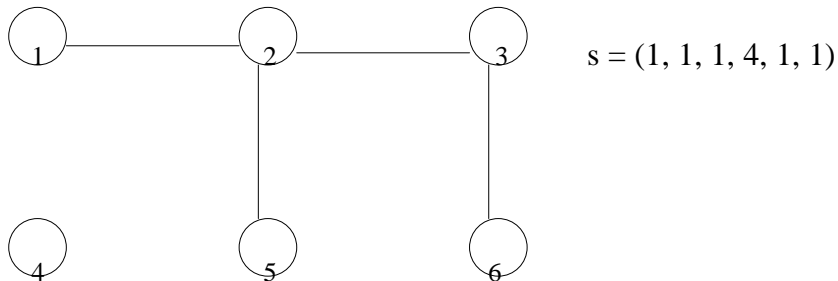
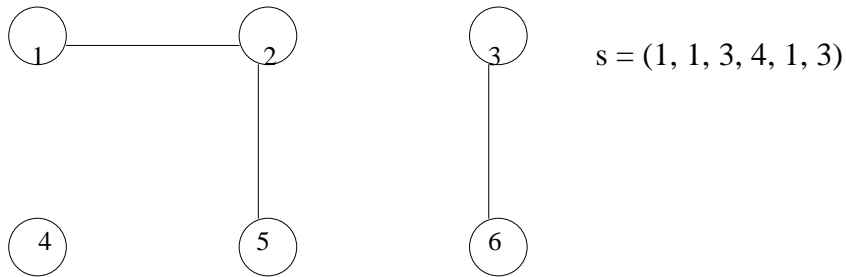
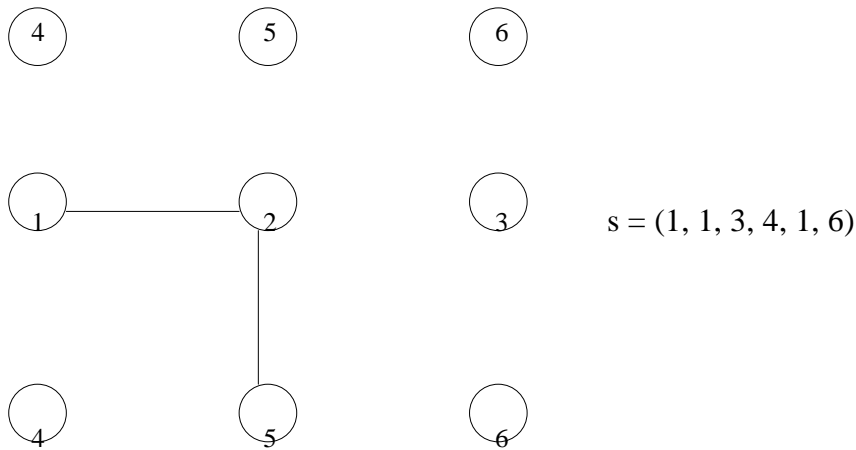


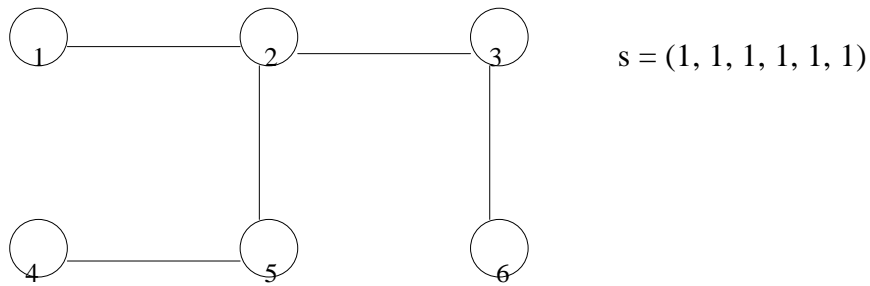
The minimum spanning tree with cost 20 is given by the following.



3.2 Kruskal's Algorithm

This method belongs to the so-called greedy paradigm. A spanning forest of the graph is a collection of trees that covers all the vertices. Sort the edges in increasing order. It is clear that the shortest edge and the second shortest edge are part of the solution. If the third shortest forms a loop, we skip it. In this way, we gradually grow a spanning forest until we finally have a single spanning tree.





The spanning forest at each stage is nothing but a collection of disjoint sets of vertices. This can be maintained by a one dimensional array s with the initial setting of $s[i] = i$ for $i = 1, \dots, n$. When we merge two disjoint sets, we adopt the name of the larger set. For example, from the 3rd stage to the 4th stage, we have the following change.

$$s = (1, 1, 3, 4, 1, 3) \Rightarrow s = (1, 1, 1, 4, 1, 1).$$

Now Kruskal's algorithm follows.

1. Sort the set of edges E in increasing order with edge cost as key into list L ;
2. **for** $i:=1$ **to** n **do** $s[i]:=i$;
3. $k:=n$; { k is the number of trees so far obtained}
4. **while** $k>1$ **do begin**
5. $(v, w) \leftarrow L$; { remove the leftmost edge from L and let it be (v, w) }
6. **if not** $(s[v] = s[w])$ **then begin**
7. $T := T \cup \{(v, w)\}$;
8. Change the set name of elements in the smaller to that of the larger;
9. **end**
10. **end**.

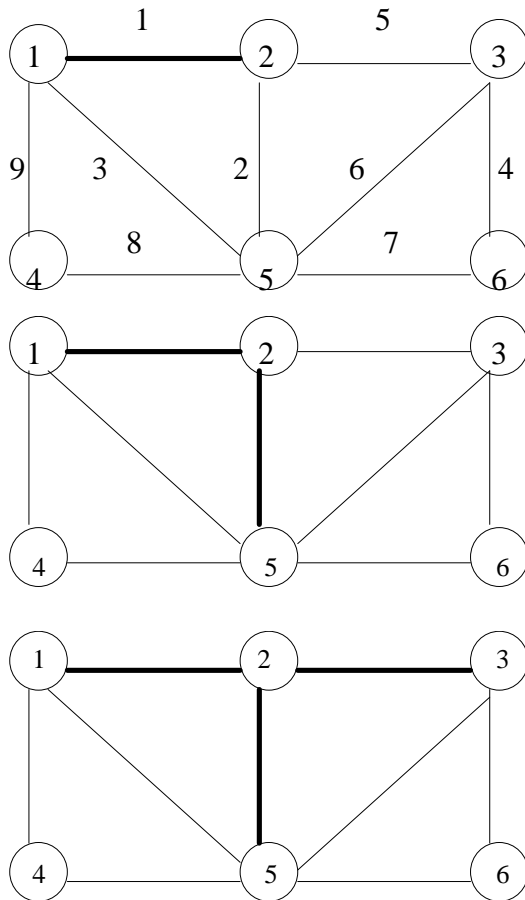
The dominant time complexity is that of sorting, which is $O(m \log m) = O(m \log n)$, since $m \leq n^2$. The overall complexity of merging two sets at line 8 is $O(n \log n)$ by the following observation. Suppose that elements $1, \dots, n$ lives in their rooms separately at the beginning, and move to larger rooms at line 8. Because they move to a larger room each time, the number of each element's moves is $\log n$. There are n elements. Thus the total cost for moving is $O(n \log n)$.

3.3 Prim's Algorithm

This algorithm is similar to Dijkstra's algorithm for shortest paths. In stead of growing a forest in Kruskal's algorithm, we grow a single tree starting from an arbitrary vertex. We maintain the frontier set F of vertices which are adjacent from the tree so far grown. Each member of F has a key which is the cost of the shortest edge to any of vertex in the tree. Each time we expand the tree, we update F and the key values of elements in F . Now Prim's algorithm follows.

1. $S := \emptyset$;
2. Put s into S ;
3. **for** v in $OUT(s)$ **do** $c[v] := L(s, v)$;
4. $F := \{v \mid (s, v) \text{ is in } E\}$;
5. **while** $|S| < n$ **do begin**
6. $v := u$ such that $c[u]$ is minimum among u in F ;
7. $F := F - \{v\}$; $S := S \cup \{v\}$;
8. **for** w in $OUT(v)$ **do**
9. **if** w is not in S **then**
10. **if** w is in F **then** $c[w] := \min\{c[w], L(v, w)\}$
11. **else begin** $c[w] := L(v, w)$; $F := F \cup \{w\}$
12. **end.**

Now the initial part of trace for our example follows with bold lines for the tree.



Similarly to Dijkstra's algorithm, if we use a binary heap for F , the time will be $O(m \log n)$ and it will be $O(m + n \log n)$ if we use a Fibonacci heap.