05 Graph (3)

College of Computer Science, CQU

Outline

- Shortest Path Problems
- Single Source Shortest Path
- Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Floyd's Algorithm

Shortest Paths Problems

- In many applications, each edge of a graph has an associated numerical value, called a **weight**. The weight of an edge is often referred to as the "cost" of the edge.

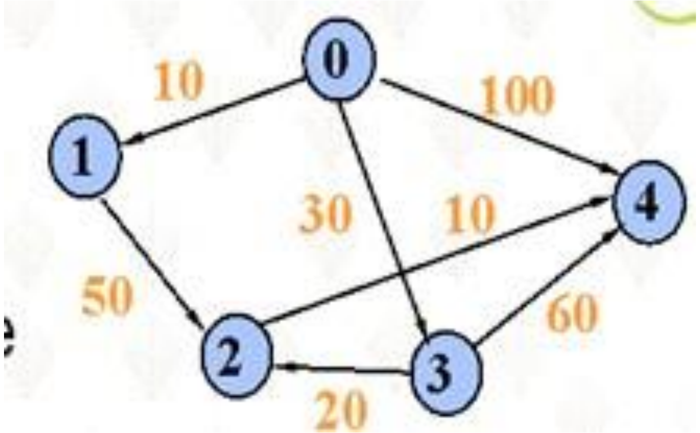
- In applications, the weight may be a measure of the length of a route, the capacity of a line, the energy required to move between locations along a route, etc.

- Usually, the edge weights are nonnegative integers.

- **Weighted graphs** may be either directed or undirected.

Shortest Paths

- **The cost of a path:** in weighted graphs, the cost of a path is the sum of the weights of its edges.
- **Shortest path**: Given two vertices A and B, there are more than one paths from A to B. The path with minimum cost is called shortest path from A to B.



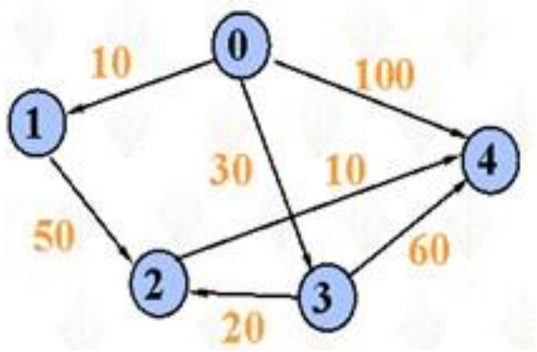
Single Source, Shortest Path

- Given a graph **G = (V, E)** and a vertex **s** $\in$ **V**, find the shortest path from s to *every* vertex in **V**
  - (Strangely, this is also typically the best way to find the shortest path from s to *any* vertex in **V**)

- Many variations:
  - weighted vs. unweighted
  - cyclic vs. acyclic
  - positive weights only vs. negative weights allowed
  - multiple weight types to optimize

Many applications!

Example



from	to	paths	lengths	S-path
0	1	(0,1)	10	(0,1)
	2	(0,1,2), (0,3,2)	60, 50	(0,3,2)
	3	(0,3)	30	(0,3)
	4	(0,4), (0,3,4), (0,1,2,4), (0,3,2,4)	100, 90, 70, 60	(0,3,2,4)

Dijkstra, Edsger Wybe

- Legendary figure in computer science;
- 1930.5.11~2002.8.6
- Supports teaching introductory computer courses without computers (pencil and paper programming)
- Supposedly wouldn't (until recently) read his e-mail; so, his staff had to print out messages and put them in his box.

Dijkstra's Algorithm for Single Source, Shortest Path

- We assume that there is a path from the source vertex v_0 to every other vertex in the graph.
- Let S be the set of vertices whose minimum distance from the source vertex has been found. Initially S contains only the source vertex.
- The algorithm is iterative, adding one vertex to S on each pass.
- We maintain an array D such that for each vertex v, D[v] is the minimum distance from the source vertex to v via vertices that are already in S.

Steps

1. let $S=\{v_0\}$, compute D[i] for each vertex v_i as following:

$$D[i]=\begin{cases} 0 & \text{if } i=0 \\ w_{0i} & \text{if } i\neq 0, \text{ and } \langle v_0, v_i\rangle \text{ is an edge, } w_{si} \text{ is the weight} \\ \infty & \text{if } i\neq 0, \text{ and } <v_s, v_i> \text{ is not an edge} \end{cases}$$

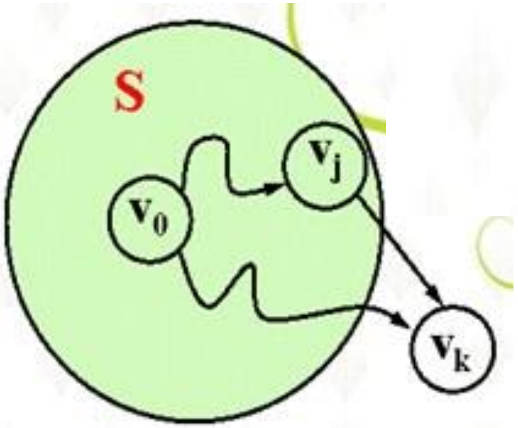
2. choose the vertex v_j such that

   $$D[j] = \min \{ D[k] \mid v_k \notin S \}$$

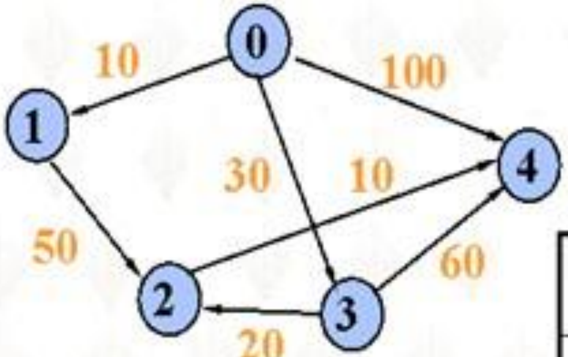
   then the v_j is terminal of the next shortest path and D[j] is its cost.
3. Place v_j in S. That is

   $$S = S \cup \{v_j\}$$
4. For each $v_k \notin S$, modify the D[k]:

   $$D[k] = \min \{ D[k],\ D[j]+weight(<v_j, v_k>)$$
5. Repeat 2---4 until all vertices have been added in S.



Dijkstra's Algorithm Trace



Steps	S	D[0]	D[1]	D[2]	D[3]	D[4]
begin	{ 0 }	0	10	∞	30	100
1	{ 0, 1 }	0	10	60	30	100
2	{ 0, 1, 3 }	0	10	50	30	90
3	{ 0, 1, 3, 2 }	0	10	50	30	60
4	{ 0, 1, 3, 2, 4 }	0	10	50	30	60

Record the Shortest paths

- The algorithm described above does not record the shortest paths. It can not output the shortest paths.
- The algorithm can be modified to record the paths by building an array pre[]. If pre[i]=k, this represents that the shortest path from v_0 to v_i is $(v_0,..,v_k,v_i)$. It is easy to prove that if $(v_0,..,v_k,v_i)$ is the shortest path from v_0 to v_i , the path $(v_0,..,v_k)$ is the shortest path from v_0 to v_k . We can output the shortest path from v_0 to v_i by outputting the shortest path from v_0 to v_k recursively and vertex to v_i
- The pre[i] is initiated by v_0. It is updated while the minimum distance is modified.

Dijkstra's Algorithm

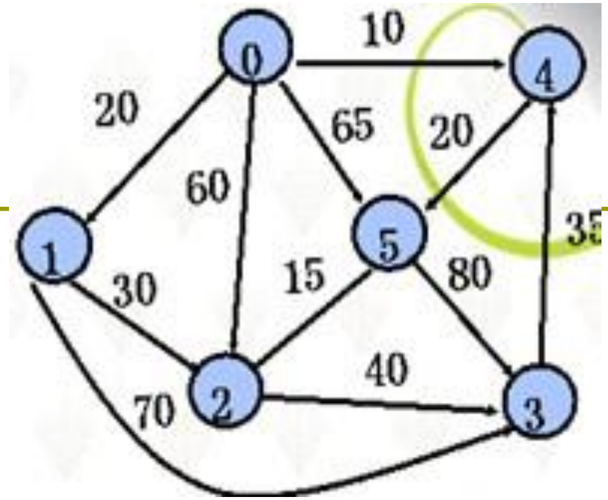
```
// Compute shortest path distances from "s".
// Return these distances in "D".
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;
  for (i=0; i<G->n(); i++) {      // Process the vertices
    v = minVertex(G, D);
    if (D[v] == INFINITY) return; // Unreachable vertices
    G->setMark(v, VISITED);
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (D[w] > (D[v] + G->weight(v, w)))
        D[w] = D[v] + G->weight(v, w);
  }
}
```

Figure 11.17 An implementation for Dijkstra's algorithm.

Dijkstra’s Algorithm

```
int minVertex(Graph* G, int* D) { // Find min cost vertex
  int i, v = -1;
  // Initialize v to some unvisited vertex
  for (i=0; i<G->n(); i++)
    if (G->getMark(i) == UNVISITED) { v = i; break; }
  for (i++; i<G->n(); i++)  // Now find smallest D value
    if ((G->getMark(i) == UNVISITED) && (D[i] < D[v]))
      v = i;
  return v;
}
```

Example



Vertices / steps		1	2	3	4	5	S
Init	D	20	60	∞	10	65	{0}
	pre	0	0	0	0	0	
1	D	20	60	∞	10	30	{0,4}
	pre	0	0	0	0	4	
2	D	20	50	90	10	30	{0,4,1}
	pre	0	1	1	0	4	
3	D	20	45	90	10	30	{0,4,1,5}
	pre	0	5	1	0	4	
4	D	20	45	85	10	30	{0,4,1,5,2}
	pre	0	5	2	0	4	
5	D	20	45	85	10	30	{0,1,2,4,3,5}
	pre	0	5	2	0	4	

Dijkstra's Algorithm（2）

- The second method is to store unprocessed vertices in a min-heap ordered by distance values.
- The next-closest vertex can be found in the heap in Θ(log n) time.
- Every time we modify **D**(*X*), we could reorder *X* in the heap by deleting and reinserting it.
- Because the objects stored on the heap need to know both their vertex number and their distance, we create a simple class for the purpose called **DijkElem**, as follows:

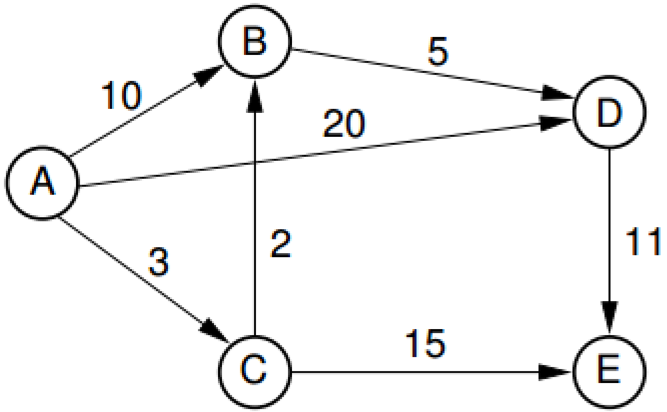
Dijkstra’s Algorithm（2）

```
class DijkElem {
  public:
    int vertex, distance;
    DijkElem() { vertex = -1; distance = -1; }
    DijkElem(int v, int d) { vertex = v; distance = d; }
};
```

Dijkstra's Algorithm

```
// Dijkstra's shortest paths algorithm with priority queue
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;              // v is current vertex
  DijkElem temp;
  DijkElem E[G->e()];       // Heap array with lots of space
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;              // Initialize heap array
  heap<DijkElem, DDComp> H(E, 1, G->e()); // Create heap
  for (i=0; i<G->n(); i++) {          // Now, get distances
    do {
      if (H.size() == 0) return; // Nothing to remove
      temp = H.removefirst();
      v = temp.vertex;
    } while (G->getMark(v) == VISITED);
    G->setMark(v, VISITED);
    if (D[v] == INFINITY) return;     // Unreachable vertices
    for (w=G->first(v); w<G->n(); w = G->next(v,w))
      if (D[w] > (D[v] + G->weight(v, w))) { // Update D
        D[w] = D[v] + G->weight(v, w);
        temp.distance = D[w]; temp.vertex = w;
        H.insert(temp);    // Insert new distance in heap
      }
  }
}
```

Dijkstra’s Algorithm



	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

All-Pairs Shortest Paths

- The problem of finding the shortest distance between all pairs of vertices in the graph called all-pairs shortest paths.
- Using the above shortest path algorithm, we can find the shortest path between all pairs of vertices, v_i and v_j, $i \neq j$.

$$\text{complexity} = n^*O(n^2) = O(n^3)$$

- Another method is **Floyd's algorithm.** It is simple and easy to implement.

Floyd’s Algorithm

- Define a K-path from vertex v to vertex u to be any path whose intermediate vertices all indices less than k.
- Define $D_k(v,u)$ to be the length of shortest k-path from vertex v to vertext u.
- $D_0(v,u)$---the weight of <v,u>
- $D_n(v,u)$---the length of shortest path from v to u
- $D_k(v,u)$ can be calculated from $D_{k-1}(v,u)$:

$$D_k(v,u) = \min \{D_{k-1}(v,u), D_{k-1}(v,k)+ D_{k-1}(k,u) \}$$

Floyd’s Algorithm

- Define a two-dimension array path[][] to record all shortest paths.
- If path[i][j]=k, the shortest path from vertex i to vertex j go through vertex k. that is, the shortest path from vertex i to vertex j is (i,..,k,..j). It is evident that (i,…,k) and (k,…,j) are the shortest path from i to k and k to j.
- We can output all shortest paths using path[][]

```
void print_allpaths( ) {
  for (int i=0; i<G->n(); i++)
    for ( int j=0; j<G->n(); j++)
      if ( i != j) {
          Printout( i) ;
          prn_pass( i, j );
          Printout(j);
      }
}
```

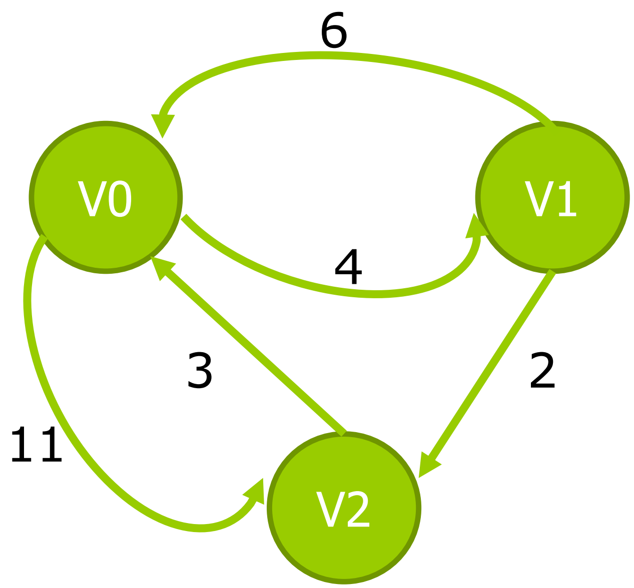
```
void prn_pass(int i, int j){
    if (path[i][j] != -1){
        prn_pass(i, path[i][j]);
        Printout( path[i][j]);
        prn_pass(path[i][j], j);
    }
}
```

Floyd’s Algorithm

```
void Floyd (Graph * G) {
   int   D[G->n()][G->n()];
    for (i = 0; i < G->n(); i++)
         for ( j = 0;  j < G->n(); j++){
              D[i][j] = G->weight(i,j);
               path[i][j] = -1;
         }
    for ( k = 0; k < G->n(); k++)
       for (i =0; i < G->n(); i++)
            for ( j = 0; j < G->n(); j++){
                 if ( D[i][k] + D[k][j] < D[i][j]){
                     D[i][j] = D[i][k] + D[k][j];
                      path[i][j] = k;
                 }
    print_allpaths();
}
```

D^{-1}

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0



D^{0}

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

D^{1}

	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

D^{2}

	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Homework

- P410, 11.9-11.11,11.16

Knowledge Points

- Chapter 11, pp.399-402

-End-