DS 6^{th} homework

58121102 Jiang Yuchu

December 7, 2022

P_{340} 5. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 6.16.

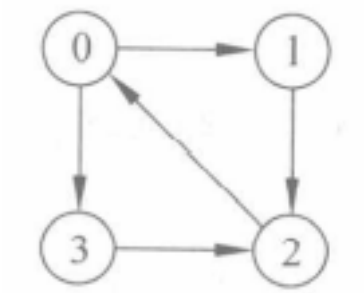


Figure 6.16 A directed graph

Answer:

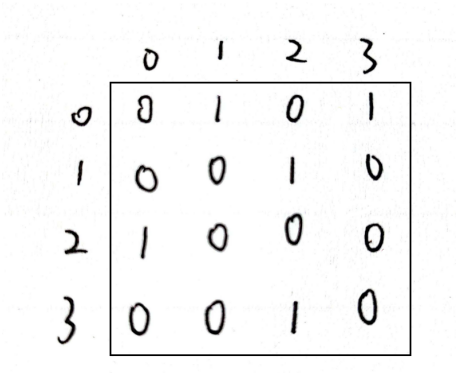


Figure 1: Adjacency-matrix representation

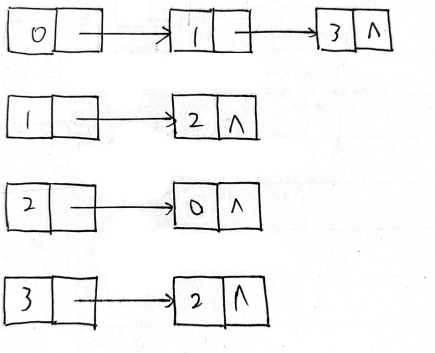


Figure 2: Adjacency-list representation

The graph given in the title is a directed graph, so it cannot be represented by an adjacency multilist. So 3 is a directed graph that I represented with a cross-linked list.

P_{340} 9. Write a C++ function to input the number of vertices and edges in an undirected graph. Next, input the edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice.What is the run time of your function as a function of the number of vertices and the number of edges?

Answer:

See *p340-9.cpp*. I tested the running time when inserting n ($n = 100, 250, 500, 1000, 2000$) edges (given in pair form, set to (a, b), then a, b belong to $[0, n]$) . Note that if either of the two vertices connected by the edge does not exist, this vertex will be added first.

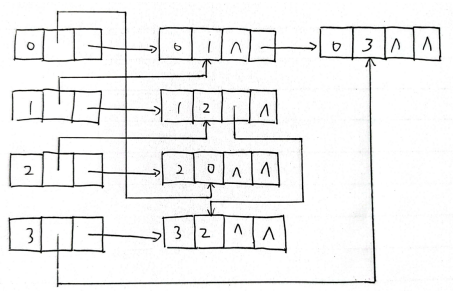


Figure 3: Adjacency-multilist representation

n	100	250	500	1000	2000
total time(us)	530.6	1127.2	1729.1	2666.6	4007.8

P_{352} 5. Write a complete C++ function for breadth-first search under the assumption that graphs are represented using adjacency lists. Test the correctness of your function using suitable graphs.

Answer:

See *p352-5.cpp*. I created a graph shown in figure 4. Then I use bfs to find vertex 7 form vertex 0. The path of its traversal is: $0 \longrightarrow 1 \longrightarrow 3 \longrightarrow 7$

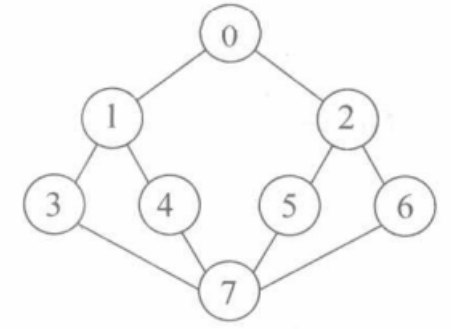


Figure 4: A undirected graph

P_{352} 6. Show how to modify function `DFS`(Program6.1), as it is used in `Components` (Program6.3), to produce a list of all newly visited vertices.

Answer:

I used the same graph mentioned in previous question. The structure of the graph is shown in figure 4. You only need to record the current parent node before entering the next layer of nodes, and then combine the parent nodes after the traversal is complete.

P_{373} 2. Let G be a directed, acyelic graph with n vertices. Assume that the vertices are numbered 0 through n-1 and that all edges are of the form $< i, j >$. where $i < j$.Assume that the graph is available as a set of adjacency lists and that each edge has a length (which may be negative) associated with it. Write a C++ function to determine the length of the shortest paths from vertex 0 to the remaining vertices. The complexity of your algorithm should be $O(n+e)$, where e is the number of edges in the graph. Show that this is the case.

Answer:

- For weighted graphs in general, we can use the Bellman-Ford algorithm to compute the single-source shortest distance in $O(VE)$ time.

- For graphs without negative weights, we can do better and use Dijkstra's algorithm to compute the single-source shortest distance in $O(E + Vlog(V))$ time.
- Can we do better for DAGs? We can compute the single-source shortest distance for a DAG in $O(V + E)$ time. The idea is to use topological sort.

See $p373-2.cpp$. The time complexity of topological sorting is $O(V+E)$. After finding the topological order, the algorithm processes all vertices, and for each vertex it performs a loop for all adjacent vertices. The total adjacent vertices in the graph are $O(E)$. So the inner loop runs $O(V + E)$ times. Therefore, the overall time complexity of the algorithm is $O(V + E)$.

P_{373} 5. Using the idea of `ShortestPath` (Program 6.8), write a C++ function to and a minimum-cost spanning tree whose worst-case time is $O(n^2)$

Answer:

See *p373-5.cpp*. The shortestpath function actually uses the dijsktra algorithm. The minimum spanning tree can be obtained by recording the path during the traversal of the algorithm. In fact, with the same idea, the Prim algorithm, Floyd algorithm and BellmanFord algorithms can all get the minimum spanning tree.

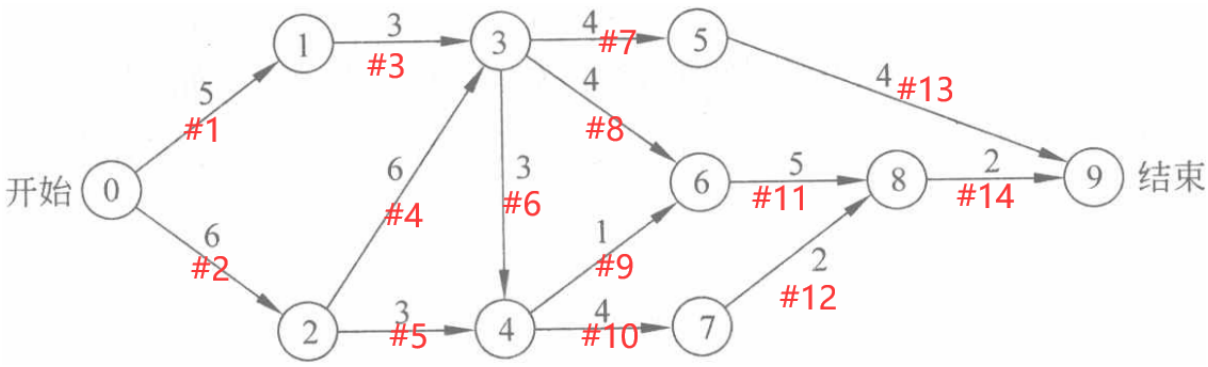
P_{389}2.

(a) For the AOE network of Figure 6.44 obtain the early, $e()$, and late, $l()$, start times for each activity. Use the forward-backward approach.

(b) What is the earliest time the project can finish?

(c) Which activities are critical?

(d) Is there any single activity whose speed-up would result in a reduction of the project length?



An AOE network

Answer:

(a).

As shown in table 1.

(b).

To complete this project, it means that event 9 needs to be completed. As can be seen from the table 2, it needs to be done by day 23 at the earliest.

(c).

As shown in table 1, if $l == e$, it means that this activity is a key activity. Activities that meet such conditions are $< 0, 2 > < 2, 3 > < 3, 4 > < 3, 6 > < 4, 6 > < 4, 7 > < 6, 8 > < 7, 8 > < 8, 9 >$.

(d).

If an activity is on all critical activity paths, accelerating such an activity can shorten the overall project duration.

activity	early time	late time	slack	critical
	e	l	l-e	l-e=0
a_1	0	4	4	No
a_2	0	0	0	Yes
a_3	5	9	4	No
a_4	6	6	0	Yes
a_5	6	12	6	No
a_6	12	12	0	Yes
a_7	12	15	3	No
a_8	12	12	0	Yes
a_9	15	15	0	Yes
a_{10}	15	15	0	Yes
a_{11}	16	16	0	Yes
a_{12}	19	19	0	Yes
a_{13}	16	19	3	No
a_{14}	21	21	0	Yes

Table 1: Early, late, and criticality values

event	early time	late time
	V_e	V_l
V_0	0	0
V_1	5	9
V_2	6	6
V_3	12	12
V_4	15	15
V_5	16	19
V_6	16	16
V_7	19	19
V_8	21	21
V_9	23	23

Table 2: Early, late time for events

P_{390}5. Define an iterator class `Topolterator` for iterating through the vertices of a directed acyclic graph in topological order.

Answer:

See *p390-5.cpp*. The difference from the previous one is that I put the implementation of the iterator into *adjacency_list.h*. In fact, the current implementation is not perfect. Here are some optimization directions that immediately come to mind:

- Since topological sequences cannot be generated forward, in order to obtain the first element, the entire topological sort must first be done. In this case, `begin()` can save the entire topological sequence. My topo order iterator only implements a forward iterator, but it can actually be implemented as a random access iterator.
- Existing iterators are invalidated after each addition or removal of an edge. In order to implement the update operation, a linked list can be formed internally for all iterators. After modifying the graph operation, each iterator is traversed to modify its internal topology sequence.

Experiment

P_{359} 1.

1 Experiment Target

Write out Kruskal's minimum-cost spanning tree algorithm (Program 6.6) as a complete program. You may use as functions the algorithms `WeightedUnion` (Program 5.22) and `CollapsingFind` (Program 5.23). Use a min-heap (Chapter 5) to select the edges in non-decreasing order by weight.

2 Basic Idea

2.1 Introduce

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

The following code is implemented with a disjoint-set data structure. Here, we represent our forest F as a set of edges, and use the disjoint-set data structure to efficiently determine whether two vertices are part of the same tree.

Algorithm 1: Kruskal

```
Input: A undirected edge-weight graph G
Output: The sum of the weights of the entire tree or a new forest F
F := ∅
foreach v ∈ G.V do
    MAKE-SET(v)
    foreach (u, v) ∈ G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F := F ∪ (u, v) ∪ (v, u)
            UNION(FIND-SET(u), FIND-SET(v))
        end
    end
end
return F
```

Using the above algorithm on the graph in Figure 5, we can finally get Figure 6.

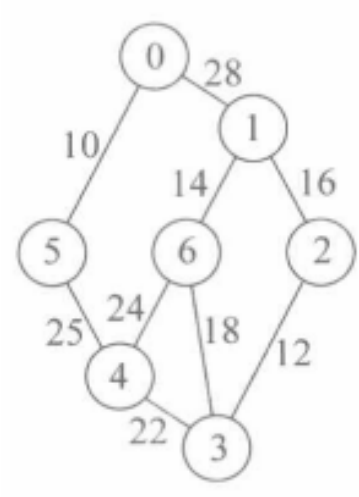


Figure 5: Original graph

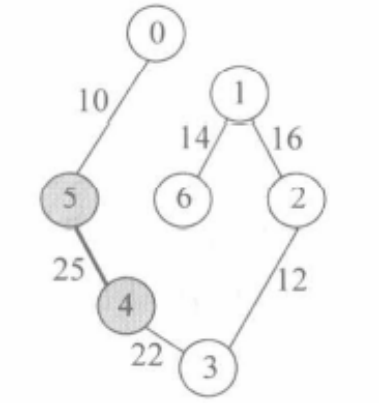


Figure 6: Minimum-cost spanning tree

2.2 Details

1. In order to take out the edge with the smallest weight every time, I first use the min-heap (that is, the priority queue) to collect all the edges.
2. `WeightedUnion` and `CollapsingFind` are not correct, so I had to implement a new version of disjoint set.

3 Procedure, Result and Analysis

I tested in figure 5. Then I obtained a undirected graph(represented as an adjacency list), which is same as 6.

```
1  vertex 0 --> 5
2  vertex 1 --> 6 --> 2
3  vertex 2 --> 3 --> 1
4  vertex 3 --> 2 --> 4
5  vertex 4 --> 3 --> 5
6  vertex 5 --> 0 --> 4
7  vertex 6 --> 1
```