DS 6^{th} homework

58121102 Jiang Yuchu

December 2, 2022

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

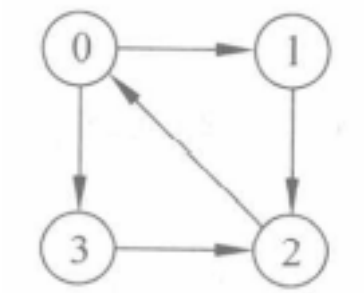

Figure 6.16 A directed graph

Answer:

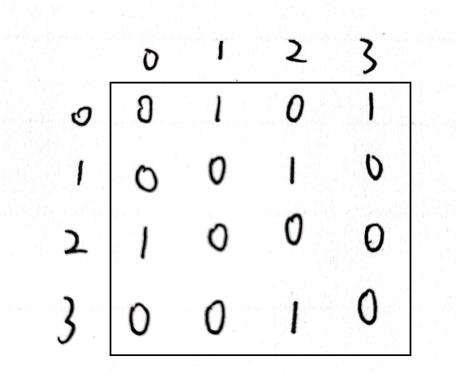

Figure 1: Adjacency-matrix representation

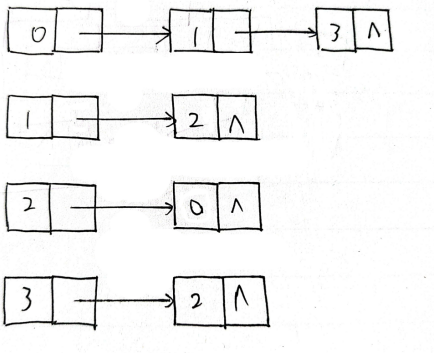

Figure 2: Adjacency-list representation

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.

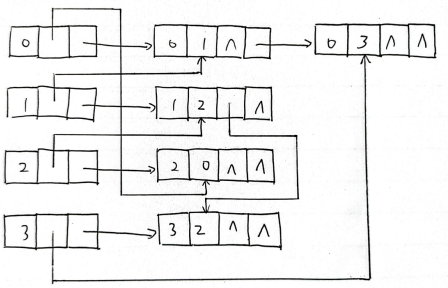

Figure 3: Adjacency-multilist representation

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: $0 \longrightarrow 1 \longrightarrow 3 \longrightarrow 7$

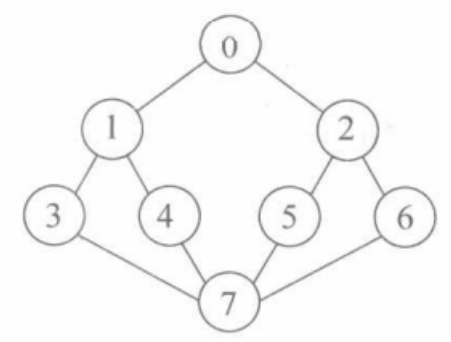

Figure 4: A undirected graph

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.

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.

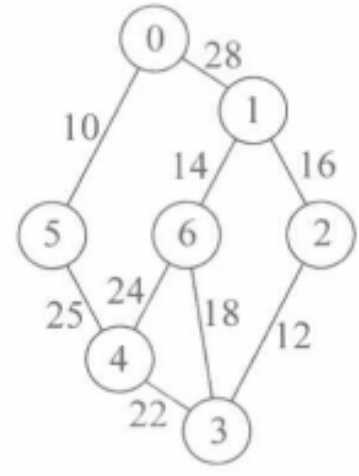

Figure 5: Original graph

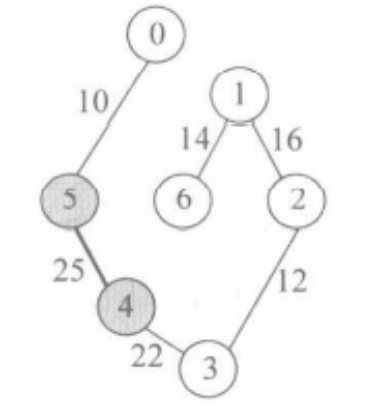

Figure 6: Minimum-cost spanning tree

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
```