Graph algorithms

Gilles Englebert

April 18, 2024

Overview

What is a graph?

A graph (V, E) is a collection of **vertices** V which are connected by **edges** E. Edges can be **directed** or **undirected** and have **weights**.

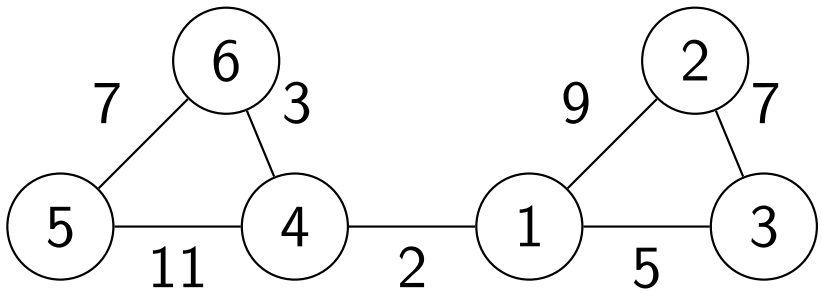


Figure: An undirected, weighted graph

In this example $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 3), (1, 3), (1, 4), (4, 5), (5, 6)\}$. The path $1 - 2 - 3 - 1$ forms a **cycle**.

Overview

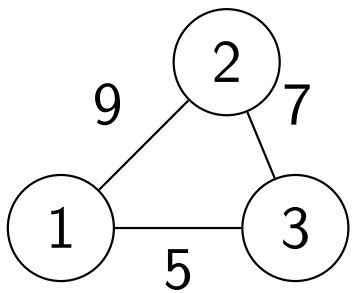


Figure: A smaller undirected, weighted graph

How should we store graphs in memory? This depends on the memory constraints and on our application. We consider three options:

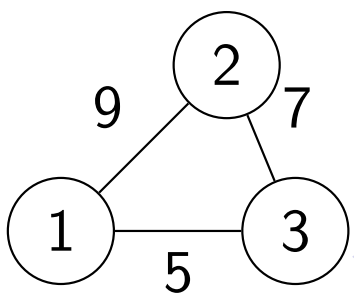
Edge list

Store the list of edges in an array of length $|E|$

```
1 //(start, end, weight)
2 int edges[V][3] = {{1, 2, 9},{2, 3, 7},{1, 3, 5}};
3
4 vector<tuple<int,int,int> > edges_vect;
5 edges_vect.push_back(tuple<int,int,int>(1, 2, 9));
6 ...
```

Memory usage: $O(|E|)$

Disadvantage: not very suitable for the algorithms we will talk about. Example: finding all the neighbours of a vertex takes $O(|E|)$ time!



Adjacency matrix

Store the distance between vertices in a $|V| \times |V|$ matrix

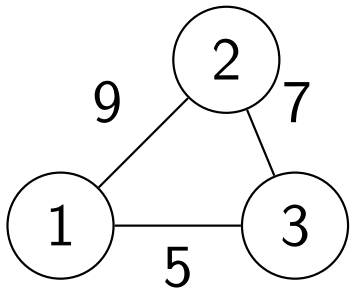
Vertices with no direct edge between them have distance $+\infty$ by definition.

```
1 int adj_mat[V][V] = {{0, 9, 5}, {9, 0, 7}, {5, 7,
      0}};
```

Memory usage: $O(|V|^2)$

Disadvantage: If the number of vertices is large, and the graph has few edges, the matrix will be mostly zeros.

$$\begin{pmatrix} 0 & 9 & 5 \\ 9 & 0 & 7 \\ 5 & 7 & 0 \end{pmatrix}$$



Neighbour list

Store a list of neighbours for each vertex.

```
1 //(neighbor ID, weight of edge)
2 vector<pair<int, int> > nb_list[v+1];
3
4 nb_list[1].push_back(pair<int, int> (2, 9));
5 nb_list[1].push_back(pair<int, int> (3, 5));
6 nb_list[2].push_back(pair<int, int> (1, 9));
7 nb_list[2].push_back(pair<int, int> (3, 7));
8 ...
```

Memory usage: $O(|E|)$

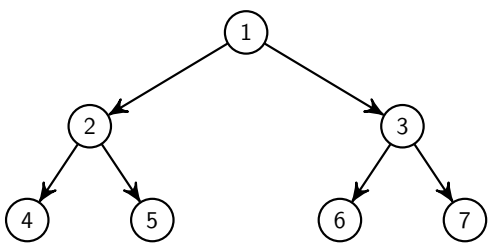
Disadvantage: Checking the distance between two vertices takes time $O(|V|)$.

Graph traversal

Problem: Given a graph $G = (V, E)$, how can I go trough all its vertices ?
There are two ways of going trough a graph, **depth first search (DFS)** and **breadth first search (BFS)**. They have different applications and are used often as part of other algorithms.

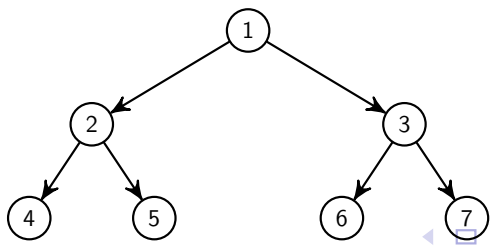
Depth first search


```
1  //(neighbor ID, weight of edge)
2  vector<pair<int, int> > nb_list[v+1];
3  bool visited[v+1] = 0;
4  void dfs(int cur) {
5      if (visited[cur])  return;
6      cout << cur <<" ";
7      visited[cur] = true;
8      for (int i; i < nb_list[cur].size(); i++)
9          dfs(nb_list[cur][i]);
10 }
11 dfs(1);
```



Breadth first search


```
1  //(neighbor ID, weight of edge)
2  vector<pair<int, int> > nb_list[v+1];
3  bool visited[v+1] = 0;
4  void bfs(int start) {
5      queue<int> q; q.push(start);
6      int cur = -1;
7      while (!q.empty()) {
8          cur = q.front(); q.pop();
9          if (visited[cur])  continue;
10         cout << cur <<" ";
11         visited[cur] = true;
12         for (int i; i < nb_list[cur].size(); i++)
13             q.push(nb_list[cur][i]);} }
14 bfs(1);
```



Connected components

Problem: Given a graph (V, E), what are its **connected components**, i.e. the subgraphs where it is possible to go in between any two vertices.

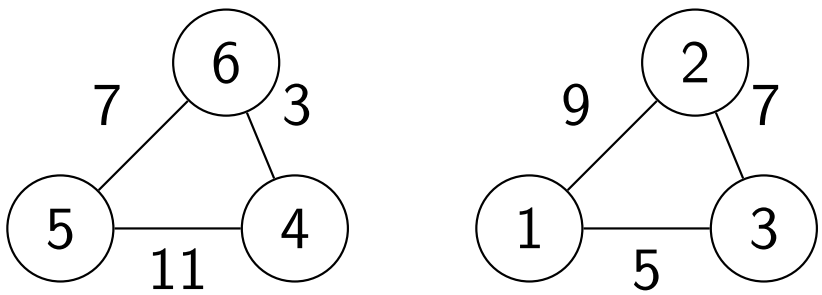


Figure: Connected components

Solution: Do either DFS or BFS at a starting vertex. All the vertices reachable are in the same connected component. Now continue doing this with unvisited vertices until all vertices are visited. **Runtime**: $O(|E| + |V|)$

Shortest path problem

Problem: Given a graph $G = (V, E)$, what is the shortest path between given vertices s (source) and t (sink) ?
To find the shortest path between two given vertices, we need to explore the entire graph. It turns out that any algorithm doing so also computes the shortest path from the source to any other vertex in the process. This is the **single source shortest path problem**. If we want to know the shortest path between any two vertices, we are solving the **all sources shortest path problem**.

Edge Relaxation

Let our source be s and consider an egde (v, w). To **relax** this edge means to check if the path from s to w via v is shorter than the currently known shortest path to w.

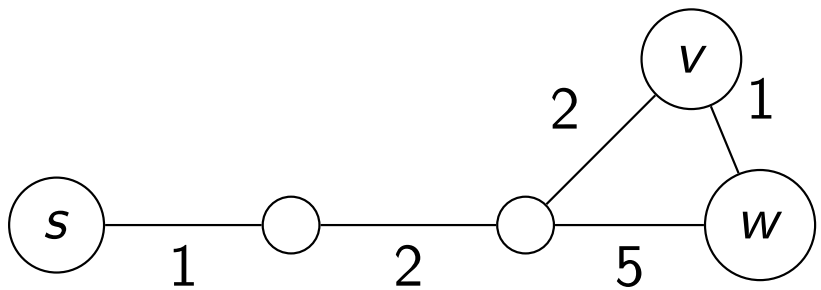


Figure: Relaxing edge (v, w)

The shortest path from s to v has length 5. The lower path from s to w has length 8. Relaxing the edge (v, w) means updating our current knowledge of the shortest paths by realising that it is faster to go through v than to take the lower edge of weight 5.

Floyd-Warshall for ASSP

We start by taking the adjacency matrix as our matrix of shortest paths (thinking of our graph as a complete graph) and relax each edge n times.

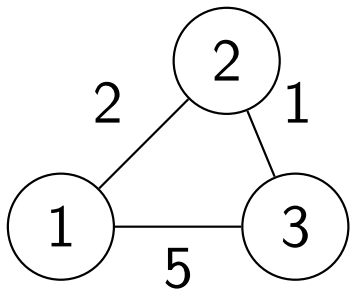


Figure: Relaxing edge (v, w)

```
1 int sh[V][V] = adj_mat;
2 for (int via = 0; via < V; via++)
3     for (int s = 0; s < V; s++)
4         for (int e = 0; e < V; e++)
5             if (sh[s][e] > sh[s][via]+sh[via][e])
6                 sh[s][e] =  sh[s][via]+sh[via][e];
```

Floyd-Warshall for ASSP

This algorithm is a bottom-up implementation of a recursive relation. Let $d_{st}^{(v)}$ be the length of the shortest path between s and t while only having intermediate vertices in $\{1, \ldots, v\}$. The following recursion holds:

$$d_{st}^{(v)} = \begin{cases} weight(s,t) & \text{if } v = 0 \\ \min(d_{st}^{(v-1)}, d_{sv}^{(v-1)} + d_{vt}^{(v-1)}) & \text{if } v \geq 1 \end{cases}$$

Now $d_{st}^{(V)}$ is the answer we are looking for. We can make our array two-dimensional instead of three-dimensional by realising that we always have $d_{st}^{(v-1)} \geq d_{st}^{(v)}$.
Runtime: $O(|V|^3)$
Memory: $O(|V|^2)$

Dijkstra for SSSP

Idea: Repeatedly relax all edges until no more relaxation can be done. Relax the most promising edges first.

```
1  //(neighbor ID, weight  of edge)
2  vector<pair <int , int > > nb[V+1];
3
4  void dijkstra(int s) {
5    priority_queue<triplet> q;
6    int dist[V] = {inf}, prec[V] = {-1};
7
8    //(distance, previous node, current node)
9    q.push(triplet(0, -1, s));
10
11   while (!q.empty()) {
12       //Remove first element in queue, see if it is
       better than the previous guess.
13       //If so, put all its neighbours in the queue.
14   }
15 }
```

Dijkstra for SSSP


```
1   while (!q.empty()) {
2     (d, prev, cur) = q.top(); q.pop();
3     if (d >= dist[cur])
4       continue;
5
6     dist[cur] = d;
7     prec[cur] = prev;
8
9     for (int i = 0; i < nb[cur].size(); i++)
10      q.push((d + nb[cur][i][1], cur, nb[cur][i
      ][0]));
11  }
```

Runtime: $O(|E| + |V| \log |V|)$ ($O(|V|^2)$ for regular queue)
Memory: $O(|E|)$

Dijkstra: Example

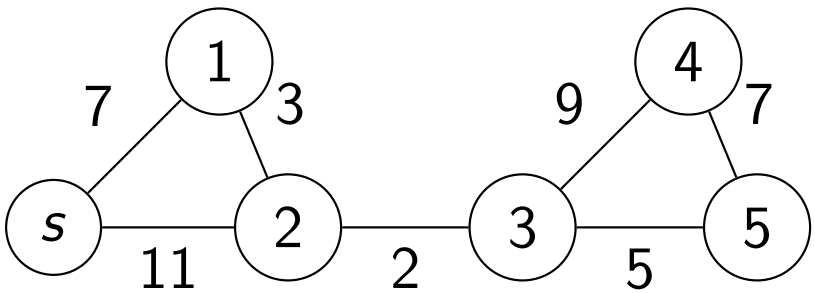


Figure: Dijkstra example, starting at s

Task: Péage

Problem: Given a graph (V, E), where some edges are designated as high-ways, find the shortest path between two nodes if you are only allowed to use one highway.

Task: Péage

Problem: Given a graph (V, E), where some edges are designated as high-ways, find the shortest path between two nodes if you are only allowed to use one highway.

Solution 1: Modify Dijkstra's algorithm to include the number of highways travelled in the data associated to a node, and act accordingly.

Task: Péage

Problem: Given a graph (V, E), where some edges are designated as high-ways, find the shortest path between two nodes if you are only allowed to use one highway.

Solution 1: Modify Dijkstra's algorithm to include the number of highways travelled in the data associated to a node, and act accordingly.

Solution 2: Double the graph so that you have two graphs G_1, G_2, and let the highways run unidirectionally from G_1 to G_2. We then look for the shortest path from the source in G_1 to the sink in G_2.

Tasks

- LIO Finals 2020: Péage
- LIO Finals 2015: Résaux
- CIL Finals 2013: Visite