Weighted graph

anhtt-fit@mail.hut.edu.vn

Weighted Graph

- We can add attributes to edges. We call the attributes weights.
  - For example if we are using the graph as a map where the vertices are the cites and the edges are highways between the cities.
  - Then if we want the shortest travel distance between cities an appropriate weight would be the road mileage.
  - If we are concerned with the dollar cost of a trip and went the cheapest trip then an appropriate weight for the edges would be the cost to travel between the cities.

Shortest Path

- Digraph $G = (V,E)$ with weight function $W: E \to R$ (assigning real values to edges)
- Weight of path $p = v_1 \to v_2 \to \dots \to v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest-Path Problems

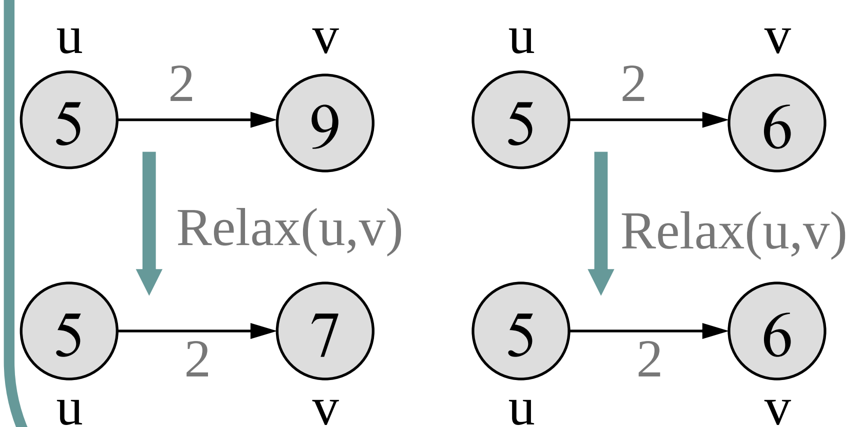
- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices. Dynamic programming algorithm.

Negative Weights and Cycles?

- Negative edges are OK, as long as there are no *negative weight cycles* (otherwise paths with arbitrary small "lengths" would be possible)
- Shortest-paths can have no cycles (otherwise we could improve them by removing cycles)
  - Any shortest-path in graph G can be no longer than $n - 1$ edges, where n is the number of vertices

Relaxation

- For each vertex *v* in the graph, we maintain *v*.**d**(), the estimate of the shortest path from *s*, initialized to ∞ at the start
- Relaxing an edge (*u*,*v*) means testing whether we can improve the shortest path to *v* found so far by going through *u*




```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v)
   then
    v.setd(u.d()+G.w(u,v))
    v.setparent(u)
```

Dijkstra's Algorithm

- Non-negative edge weights
- Like breadth-first search (if all weights = 1, one can simply use BFS)
- Use *Q*, a priority queue ADT keyed by *v*.**d**() (BFS used FIFO queue, here we use a PQ, which is re-organized whenever some **d** decreases)
- Basic idea
  - maintain a set *S* of solved vertices
  - at each step select "closest" vertex *u*, add it to *S*, and relax all edges from *u*

Demo

- demo-dijkstra.ppt

Dijkstra’s Pseudo Code

- Input: Graph *G*, start vertex *s*

```
Dijkstra(G,s)
for each vertex u ∈ G.V()
   u.setd(∞)
   u.setparent(NIL)
s.setd(0)
// Set S is used to explain the algorithm
Q.init(G.V())  // Q is a priority queue ADT
while not Q.isEmpty()
   u ← Q.extractMin()

   for each v ∈ u.adjacent() do
      Relax(u, v, G)
      Q.modifyKey(v)
```

relaxing edges

Implementation

- Modify the graph API to support weighted edges as the following

```
#define INFINITIVE_VALUE  10000000
typedef struct {
   JRB edges;
   JRB vertices;
} Graph;
void addEdge(Graph graph, int v1, int v2, double weight);
double getEdgeValue(Graph graph, int v1, int v2); // return
    INFINITIVE_VALUE if no edge between v1 and v2
int indegree(Graph graph, int v, int* output);
int outdegree(Graph graph, int v, int* output);
double shortestPath(Graph graph, int s, int t, int* path,
    int*length); // return the total weight of the path and the path is
    given via path and its length. Return INFINITIVE_VALUE if no
    path is found
```

Quiz

- Write the implementation of the weighted graph API. Test the API using the following example

```
Graph g = createGraph();
// add the vertices and the edges of the graph here
int s, t, length, path[1000];
double weight = shortestPath(g, s, t, path, &length);
if (weight == INFINITIVE_VALUE)
   printf(“No path between %d and %d\n”, s, t);
else {
   printf(“Path between %d and %d:”, s, t);
   for (i=0; i<length; i++) printf(“%4d”, path[i]);
   printf(“Total weight: %f”, weight);
}
```

Solution

- dijkstra.c

Mini Project II

- The objective of this project is to simulate a bus map in Hanoi.
- Firstly, you have to collect data about Hanoi's bus map in the form of a graph where
  - Each vertex is a bus station corresponding to a place in Hanoi
  - The edges connect the bus stations via the bus lines.
  - E.g., There are 16 stations connected by bus No 1A: "Yên Phụ - Hàng Đậu - Hàng Cót - Hàng Gà - Hàng Điếu - Đường Thành - Phủ Doãn - Triệu Quốc Đạt - Hai Bà Trưng - Lê Duẩn - Khâm Thiên - Nguyễn Lương Bằng- Tây Sơn - Nguyễn Trãi - Trần Phú (Hà Đông) - Bến xe Hà Đông"
  - Cf., http://www.hanoibus.com.vn/InfobusVN/hanoibus/index.asp?pPage=lotrinh.htm

Mini project II (cont.)

  - Each edge in the graph marked with the bus lines which traverse from one to the other. E.g., The edge “Yên Phụ - Trần Nhật Duật” is marked with 4A, 10A.
- Organize and store the data in a file to be loaded in the program when running
- Rewrite the graph API to be able to store the bus map in memory
- Develop a functionality to find the “shortest path” to move from a place to another. E.g., From “Yên Phụ” to “Ngô Quyền”.