AlgoLab Reference Sheet

Emilien Pilloud

January 2019

1 STL Usefull Functions & Snippets

2 BGL Theory

2.1 Shortest Paths

2.1.1 Find the shortest path from a to b

Use Dijktra's algorithm which runs in $O(VlogV + E)$.

2.1.2 Find all shortest paths from a to b

Use Dijktra's twice: once from a to b, once from b to a. Remember the shortest length, iterate over the vertices v and see of d(v, a) + d(v, b) = min_d.

```
#include <boost/graph/dijkstra_shortest_paths.hpp>
//typedef ...
Graph G(N);
std::vector<uint64_t> dists_a(n, 0), dists_b(n, 0);
boost::dikstra_shortest_paths(G, a, boost::predecessor_map(&dist_a[0]));
boost::dikstra_shortest_paths(G, a, boost::predecessor_map(&dist_a[0]));
uint64_t min_d = dists_a[b];
BGL_FORALL_EDGES(e, G, Graph){
    Vertex from = boost::source(e, G);
    Vertex to = boost::target(e, G);

    if(dists_a[from] + weights[e] + dists_b[to] == min_d)
        //this edge is part of a shortest path
}
```

c.f. Marathon

2.1.3 Find the shortest path that minimizes the number of edges used

Modify the Dijkstra priority queue to have a double ordering over the path length and the number of edges used

```
struct S{
    uint64_t lenght, n_edges, v;
    const bool operator<(const S& s2) const{
        return std::make_pair(length, n_edges) > std::make_pair(s2.length, s2.n_edges);
    }
};
std::priority_queue<S> Q;
// Dijktra ...
```

2.1.4 Find all pair of shortest paths

If sparse, e.g. tree, use johnson_shortest_path algorithm from BGL.

2.2 Maximum Flows

2.2.1 Push relabel maximum flow

The push relabel maximum flow runs in $O(V^3)$ but is in practice the best max flow algorithm.

2.2.2 Flow with demand on the edges

This problem is a max flow problem where you want to enforce a minimum quantity d_i of flow to go throw given edges. This can be done by "teleporting" flow from a to b, i.e. take the d_i units of flow from the u_i vertex and add d_i unit of flow in the v_i vertex. You must be careful not to have negative values on given vertices. This is especially easy to do if each node has a demand and/or a supply.
c.f. Kingdom Defense

2.2.3 Flow with maximal vertex capacity

This can be solved by splitting a vertex u_i with a capacity constraint into two nodes: u_i^{in} and u_i^{out}. Then just add an edge between the two with the wanted flow constraint.

2.2.4 Conditional edge (OR gate)

It may be useful to have edges in a flow graph to where the flow can go either fom a to b or from b to a. This can be modelled by this construction (see latex comment):

2.2.5 Edge Disjoint Paths from a to b

If the Graph is directed, this can be easily reformulated as a flow problem with edges of capacity one and then finding the maximum flow.
If the graph is undirected, we can transform the undirected edges into two directed edge and compute the flow. We notice that if there's a unit of flow $u \to v$ and $v \to u$, then we can erase the flow from both edges and the amount of flow is conserved. In the case where you want to get the edges from the disjoint paths, one should care about this issue.

2.2.6 Minimum Cut from a to b

We know from the minimum cut theorem that a minimum cut is equal to the maximum flow of the graph.

2.2.7 Find the overall minimum cut

It is sufficient to compute all the minimum cuts from a fixed vertex a to all other vertices b. Indeed, it can be shown that in the overall minimum cut, if a and b are in different cutting sets, then the maximum flow between them is the overall minimum cut.
To explicitly get the two disjoint sets of vertices, run a BFS from the source of the max_flow.
Note: this could maybe be solved with the *stoer_wagner_min_cut* in $O(VE + V^2 logV)$ from BGL library. (needs to be checked)
c.f. Algocoon

2.2.8 Minimum cut at the vertices

The minimal cut problem can sometimes be need on the vertices, i.e. what is the smaller number of vertices such that the flow going throw them is maximized. This can also be seen as the set of vertices which is the "bottleneck". This can efficiently be solved by splitting all vertices in *in* and *out* vertices with capacity one (or more depending on the problem).
c.f. supplementary_exercices/phantom_menace

2.2.9 Vertex Cover for bipartite graphs

Vertex cover can be efficiently computer in a bipartite graph by: Computing the max flow and finding the set of vertices reachable from the source in the residual graph.
c.f. Satellites

2.2.10 Best matching in bipartite graphs

This can also be solved using (regular or min cost) max flows. You need to create the bipartite graph dependent on the problem and then find the maximum flow with the appropriate algorithm.

2.3 Connected Components

If the graph G is undirected, used *connected_components*:

```
std::vector<int> component(num_vertices(G));
int num = connected_components(G, &component[0]);
```

If the graph G is directed, used *strong_components*.
There is also the notion of biconnected components which can give you the articulation points of the graph and hence allows to find the edges essential to the connected property. (Biconnected component of size 2)
c.f. Important bridges

To efficiently compute incremental components for instance with restriction on the edges, one can use either the incremental component function of BGL or quite quickly implement it using union-find structure. This can be done like this:

```
#include <boost/pending/disjoint_sets.hpp>
typedef boost::disjoint_sets_with_storage<> Uf;
//...
// union op
Uf.union_set(int elem_from_a, int elem_from_b);
// link (union of representative)
Uf.link(int rep_a, int rep_b);
// find op
Uf.find_set(int elem);
```

2.4 MST

The minimum spanning tree can be computer via prim's or kruskal with the following methods from BGL ($O(ElogE)$):

```
std::vector<Edge> MST;
boost::kruskal_minimum_spanning_tree(g, std::back_inserter(spanning_tree));
```

2.4.1 Compute the closest MST

It is easy to show that the closest MST in term of its sum of weights is at most one edge away from the MST. Hence, compute in $O(V^2)$ all the max edge from the nodes via BFS. And then for all edges not in the MST, compute the added difference if this edge was added instead in the MST (The max edge from u to v would be replaced by this given edge).

2.5 Maximal matching on arbitrary graph

A maximal matching on an arbitrary graph can be computed in $O(mn\alpha(m,n))$ where α is at most 4 for all feasible input. In the context of AlgoLab, use the checked version to avoid all potential problems. Example:

```
Graph g; std::vector<Vertex> mate(n);
bool success = checked_edmonds_maximum_cardinality_matching(g, &mate[0]);
assert(success);
```

In most problems, this algorithm is too expensive and one should think about a bipartite representation to compute the maximal matching efficiently. c.f. Buddies

3 BGL Usefull Functions & Snippets

3.1 BGL Graph utility

This file allows to easily print a graph. Saves a lot of debuging time.

```
#include <boost/graph/graph_utility.hpp>
// Compact way of printing the graph adjacency list
// recall: should use the std::cerr as output to avoid getting wrong-answers
// from the judge whereas it is only a forgotten print.
boost::print_graph(G, std::cerr);

// It is also possible to print edges
// But I don't find this so useful in most cases
```

But there's way more, including many useful predicates:

```
bool is_adjacent(Graph& g, Vertex a, Vertex b){..}
```

3.2 BGL Iteration Macros

Those predefined macros save a lot of time when you need to iterate of vertices or edges from a BGL graph.

```
// Needed additional import
#include <boost/graph/iteration_macros.hpp>

Graph G(n);
// Iterate over all vertices
BGL_FORALL_VERTICES(v, G, Graph){..}
// Iterate over all edges
BGL_FORALL_EDGES(e, G, Graph){..}
// VERY USEFUL : Iterate over outgoing edges
// A INEDGES variant also exists
BGL_FORALL_OUTEDGES(from_vertex, e, G, Graph){..}
// Iterate over adjacent vertices
BGL_FORALL_ADJ(from_vertex, v, G, Graph){..}
```

4 CGAL Usefull Functions & Snippets

```
    // Inexact Kernel
    #include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
    // Exact Kernel
    #include <CGAL/Exact_predicates_exact_constructions_kernel.h>
    // Exact Kernel with exact sqrt
    #include <CGAL/Exact_predicates_exact_constructions_kernel_with_sqrt.h>
```

4.1 Useful Predicates

Try to use predicates only before moving on to constructions. Predicate-only based code is much faster.

```
//left/right turns. check if the third point z is left or right of the vector xy
bool leftturn(P x, P y, P z){..}
bool rightturn(P x, P y, P z){..}

// Check if the fourth point z lies in the circle defined by w, x, y.
bool incircle(Pw, P x, P y, P z){..}

// Compared the distances between points
// True if d(p, q) < d(p, r)
bool CGAL::has_smaller_distance_to_point(P p, P q, P r){..}

// Same for larger predicate
bool CGAL::has_larger_distance_to_point(P p, P q, P r){..}
```

4.2 Triangulation

4.2.1 General Info

Delaunay Triangulation have the following properties:

- Empty Circle prop: Take three points from the triangulation and construct a circle from it. No other points in the triangulation will be in its interior.
- It minimizes the angles

- It contains the MST

Some cases where the triangulation is useful:

- Find a nearest vertex in $\mathcal{O}(nlog(n))$ in the number of points in the triangulation.
- Find a set of 3*n edges containing the MST of a cloud of points.
- Optimal Movement under constraints related to the proximity of the vertices. cf. Hongkong, H1N1, Graypes...

4.2.2 Includes & Typedef

```
// Basic includes
#include <CGAL/Delaunay_triangulation_2.h>
typedef CGAL::Delaunay_triangulation_2<K> Tr;
// Vertex with infos
typedef CGAL::Triangulation_vertex_base_with_info_2<L, K> Vb;
typedef CGAL::Triangulation_data_structure_2<Vb> Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds> Tr;
```

4.2.3 Useful Getters and functions

```
// Get a point from a Vertex_handle
Vertex_handle vh;
P p = vh->point();

// Find a the nearest vertex in the triangulation from a general point
Vertex_handle nearest = tr.nearest_vertex(p);

// Find the face in which a point lies
Face_handle f = tr.locate(p);

// Random access to neighboring faces
Face_handle neighbor_face_i = f->neighbor(i%3);

// Random access to vertex from face
Vertex_handle v_i = f->vertex(i%3);

// The edge from current face f to neighbor i is by def the edge btw
// f->vertex((i+1)%3) and f->vertex((i+2)%3)

// Get the Voronoi diagram center, warning: it is a construction
K::Point_2 center = tr.dual(f);

// is_infinite allows to check for infinite Faces and Edges in the triangulation.
// Usually, those are special cases you want to treat differently in your programs.
Tr::is_infinite()
```

4.2.4 Adding vertices to the triangulation

```
    Tr tr; tr.insert(p);
    // But it is always better to add all points at once if possible
    // CGAL will optimize the order of insertion
    tr.insert(pts.begin(), pts.end());
```

4.2.5 Conversion from Triangulation to CGAL shapes

```
Tr::Face_handle F_h;
// Much easier than getting vertices by hand
Tr::Triangle_2 triangle = tr.triangle(F_h);

// Edge to segment
// recall: Edge are std::pair<Tr::Face_handle, int> with 0 <= int < 3
Tr::Edge e;
K::Segement_2 s = tr.segment(e);
```

4.2.6 Vertex and Faces Infos

Add info structure to the vertex type. Can be useful to remember an ID or more data related to the nodes in the triangulation.

```
// Vertex with infos
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Triangulation_face_base_with_info_2.h>
struct L{
    // ...
};
struct N{
    // ...
};
typedef CGAL::Triangulation_vertex_base_with_info_2<L, K> Vb;
typedef CGAL::Triangulation_face_base_with_info_2<N, K> Fb;
typedef CGAL::Triangulation_data_structure_2<Vb, Fb> Tds;
typedef CGAL::Delaunay_triangulation_2<K, Tds> Tr;

// Can be accessed via the info() method
Tr::Vertex_handle v;
auto infos = v->infos();

// To push a point, it needs to be a std::pair<K::Point_2, L>
tr.insert(std::make_pair(p, infos));
```

4.2.7 Iterators

```
// iterate over all faces
auto itr = tr.all_faces_begin();
// iterate only over finite faces
auto itr = tr.finite_faces_begin();
// Circulator over the incident edges
auto circ = tr.incident_edges(vertex_handle)
do{
    // ...
    circ++;
}while(circ != tr.incident_edges(vertex_handle));

// Circulator over the incident faces
auto circ = tr.incident_faces(vertex_handle);
do{
    // ...
    circ++;
}while(circ != tr.incident_faces(vertex_handle));

// Useful Special Case: Get all infinite faces
auto infinite_faces_circ = tr.incident_faces(tr.infinite_vertex());
```