A brief introduction to pb_ds for ICPC

Yuchen Lei

July 14, 2019

Abstract

pb_ds, also known as Policy-Based Data Structures, is a g++ specific library of policy-based elementary data structures. In the rest of this article, we will take a glance at this library.

Contents

Heap

In particular C++, when we need a heap, we will include queue and just use priority_queue. In most situations, naive priority_queue does fit in use: $O(\log n)$ push and pop, and $O(1)$ top operation. But in the most tough situation (for example data maker is yswang), we may need faster heap to accomplish such mission.

Lots of different kinds of heap are provided in pb_ds, in order to use them, just include `ext/pb_ds/priority_queue.hpp` and use `__gnu_pbds::priority_queue`:

```
template<
    typename Value_Type,
    typename Cmp_Fn = std::less<Value_Type>,
    typename Tag = pairing_heap_tag,
    typename Allocator = std::allocator<char> >
class priority_queue;
```

The biggest different is the `Tag` template argument, pb_ds use this argument to determine which kind of head to use actually. As for now, we can select `Tag` from `pairing_heap_tag`, `binary_heap_tag`, `binomial_heap_tag`, `rc_binomial_heap_tag`, or `thin_heap_tag`. Here is a simple table which compares these heaps.

	Push	Pop	Modify/Erase	Join
pairing_heap_tag	$O(1)$	$O(\log n)/O(n)$	$O(\log n)/O(n)$	$O(1)$
binary_heap_tag	$O(\log n)/O(n)$	$O(\log n)/O(n)$	$O(1)$	$O(1)$
binomial_heap_tag	$O(1)/O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
rc_binomial_heap_tag	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
thin_heap_tag	$O(1)$	$O(\log n)/O(n)$	$O(\log n)$	$O(\log n)/O(n)$

Unlike STL's implementation, priority_queue in pb_ds supports more opeartions such as remove, modify and join.

For example, assuming that we are implementing heap optimized dijkstra to find single source shortest path, we need to update shortest path length for all points which will be relaxed. Traditionally, we will push a new `{new_dis, point}` into heap so we can find correct shortest point right now and when the top element's distance doesn't match the global answer, we can directly drop it for it is a obsoleted element. This can cause memory overhead obviously.

But with the modify operation of pb_ds's priority_queue, we can elegantly solve this problem, we only need to add a global array to store all the iterators of points.

A g++ offical example snippet is shown below.

```
typedef std::pair<size_t, size_t> pq_value;
struct pq_value_cmp : public binary_function<pq_value, pq_value, bool>
{
  inline bool
  operator()(const pq_value& r_lhs, const pq_value& r_rhs) const
  { return r_rhs.second < r_lhs.second; }
};
typedef __gnu_pbds::priority_queue< pq_value, pq_value_cmp> pq_t;
vector<pq_t::point_iterator> a_it;
for (size_t i = 0; i < num_vertices; ++i)
    a_it.push_back(p.push(pq_value(i, graph_inf)));
p.modify(a_it[0], pq_value(0, 0));
```

```
if (pot_dist < a_it[neighbor_i]->second)
        p.modify(a_it[neighbor_i], pq_value(neighbor_i, pot_dist));
```

Here's another (easier) example to solve HDU2544:

```
#include <bits/stdc++.h>
#include <ext/pb_ds/priority_queue.hpp>
using namespace std;
const int N = 105, M = 20500;
int adj[N], nxt[M], to[M], len[M], ecnt;
int dis[N];
inline void addEdge(int f, int t, int l)
{
    ecnt++;
    nxt[ecnt] = adj[f];
    adj[f] = ecnt;
    to[ecnt] = t;
    len[ecnt] = l;
}
struct node
{
    int u, l;
    bool operator<(const node &rhs) const noexcept { return l > rhs.l; }
};
typedef __gnu_pbds::priority_queue<node> heap;
heap::point_iterator ite[N];
int main()
{
    heap H;
    for (int n, m; scanf("%d%d", &n, &m), n | m;)
    {
        H.clear();
        ecnt = 0;
        memset(adj, 0, sizeof adj);
        memset(dis, 0x3f, sizeof dis);
        for (int i = 0; i < m; i++)
        {
            int x, y, z;
            scanf("%d%d%d", &x, &y, &z);
            addEdge(x, y, z);
            addEdge(y, x, z);
        }
        for (int i = 1; i <= n; i++) ite[i] = H.push({i, *dis});
        for (H.modify(ite[1], {1, dis[1] = 0}); !H.empty(); H.pop())
            for (int u = H.top().u, e = adj[u]; e; e = nxt[e])
                if (dis[to[e]] > dis[u] + len[e])
                    H.modify(ite[to[e]], {to[e], dis[to[e]] = dis[u] + len[e]});
        printf("%d\n", dis[n]);
    }
    return 0;
}
```