Document for DD's CH code

潘睿

目录

- 背景
- CH 层次图构建
- CH 查询最短路

背景

背景

- CH 的层次图构建
  - 预处理
    - 确定一个**缩点的顺序**
    - 每次按这个顺序选一个点 u 缩
      - 如果 v→u→w **可能是 v 到 w 的唯一的最短路**，在原图中增加 shortcut 捷径 (v,w) ，权值为 (v,u) 和 (u,w) 的权值和
    - 删除点 u

Algorithm 1: SimplifiedConstructionProcedure($G = (V, E), <$)

```
1 foreach u ∈ V ordered by < ascending do
2     foreach (v, u) ∈ E with v > u do
3         foreach (u, w) ∈ E with w > u do
4             if ⟨v, u, w⟩ "may be" the only shortest path from v to w then
5                 E := E ∪ {(v, w)} (use weight w(v, w) := w(v, u) + w(u, w))
```

背景

- CH 的层次图构建
  - 预处理
    - 确定一个**缩点的顺序 (terms? 怎么 update ？)**
    - 每次按这个顺序选一个点 u 缩
      - 如果 v→u→w **可能是 v 到 w 的唯一的最短路**，在原图中增加 shortcut 捷径 (v,w) ，权值为 (v,u) 和 (u,w) 的权值和
    - 删除点 u

Algorithm 1: SimplifiedConstructionProcedure($G = (V, E), <$)

```
1 foreach u ∈ V ordered by < ascending do
2     foreach (v, u) ∈ E with v > u do
3         foreach (u, w) ∈ E with w > u do
4             if ⟨v, u, w⟩ "may be" the only shortest path from v to w then
5                 E := E ∪ {(v, w)} (use weight w(v, w) := w(v, u) + w(u, w))
```

背景

- Node order（缩点顺序）
  - Edge difference（边数差）
  - Cost of contraction（缩点代价）
  - Uniformity（均匀性）
  - Cost of queries（查询代价）
  - Global measures（全局特征）

背景

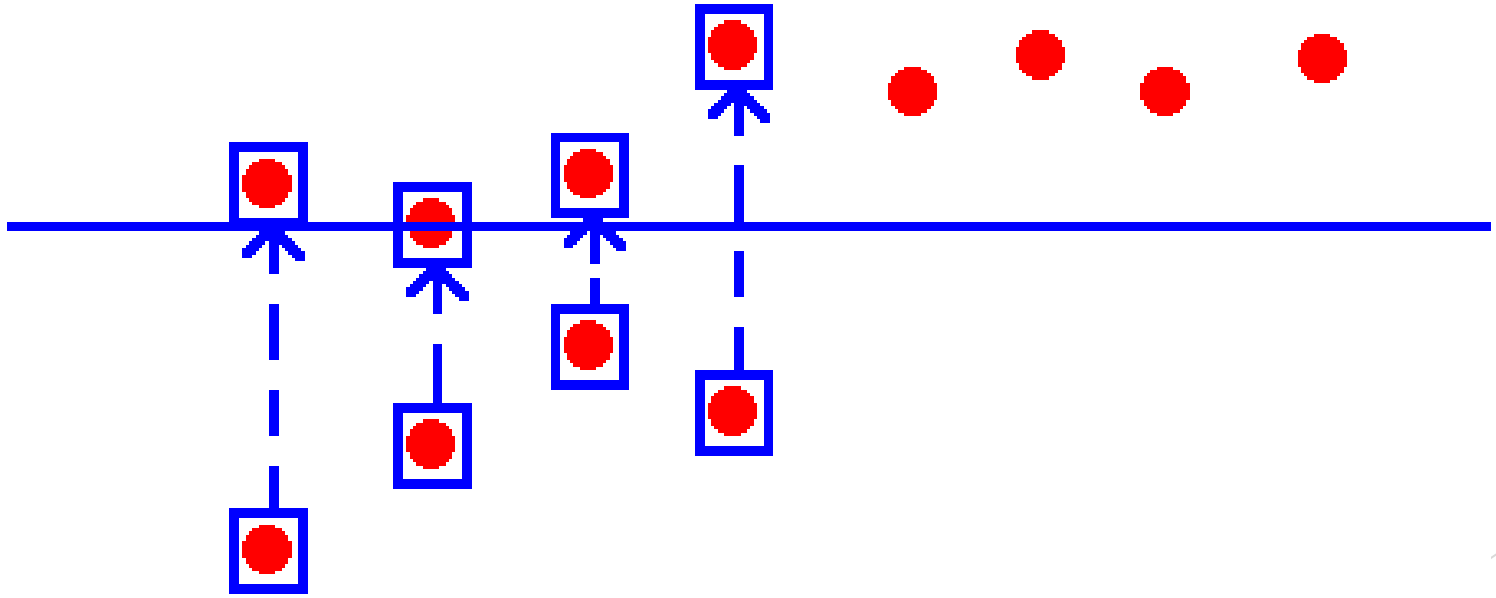
- Node order (缩点顺序)
  - 每一个 term 的数值在预处理过程中是固定的吗？
  - 怎么办？
  - 答：
    - 周期性全局 update
    - Contract 一个点后 update 邻居点
    - Lazy update

背景

- Node order (缩点顺序)
  - 每一个 term 的数值在预处理过程中是固定的吗？
    - Lazy update
      - 1. 使用一个优先队列维护所有当前未被删除的点
      - 2. 更新队列头
      - 3. 如果其还是队列头那么该点就是当前的最小点
      - 4. 否则更换队列头，跳到 2
    - 对于权值随着 contraction 单调上升的情况是正确的

背景

- Node order (缩点顺序)
  - 每一个 term 的数值在预处理过程中是固定的吗？
    - Lazy update



背景

- CH 的层次图构建
  - 预处理
    - 确定一个**缩点的顺序**
    - 每次按这个顺序选一个点 u 缩
      - 如果 v→u→w **可能是 v 到 w 的唯一的最短路 (局部搜索用了哪些优化？)** ，在原图中增加 shortcut 捷径 (v,w) ，权值为 (v,u) 和 (u,w) 的权值和
    - 删除点 u

Algorithm 1: SimplifiedConstructionProcedure($G = (V, E), <$)

```
1 foreach u ∈ V ordered by < ascending do
2     foreach (v, u) ∈ E with v > u do
3         foreach (u, w) ∈ E with w > u do
4             if ⟨v, u, w⟩ "may be" the only shortest path from v to w then
5                 E := E ∪ {(v, w)} (use weight w(v, w) := w(v, u) + w(u, w))
```

背景

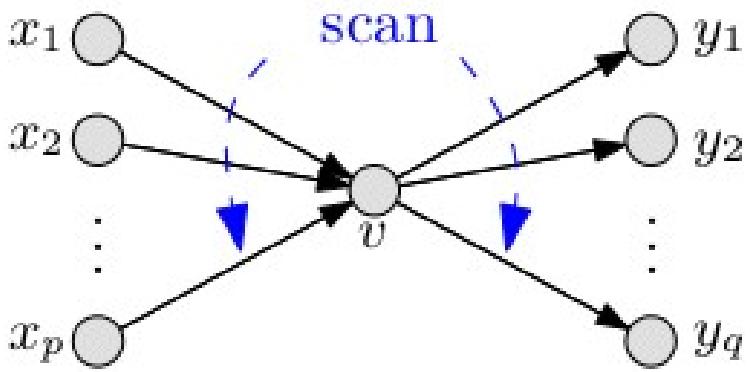
- Witness path search（确定 v->u->w 是不是 v 到 w 唯一的最短路）
  - 预处理需要找很多次 witness path ， Dijkstra 太慢怎么办？
    - 限制搜索范围的几种方法
      - 限制搜索点数
      - 限制搜索深度

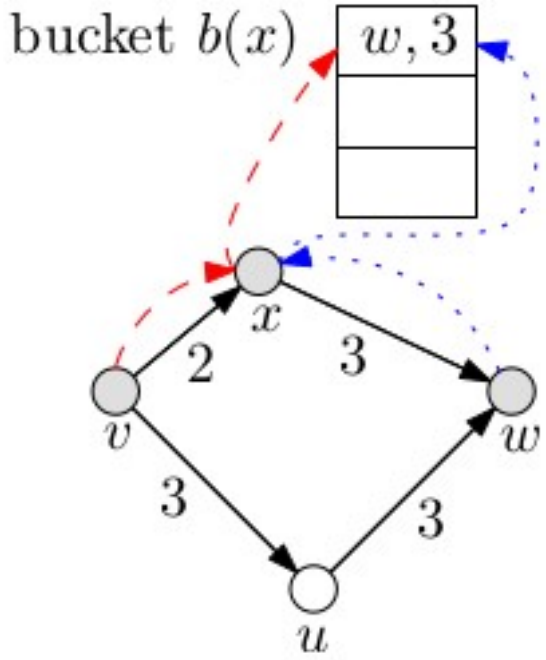
Limit the number of settled nodes. A local search, implemented as a modified Dijkstra algorithm, can be stopped after a certain number of nodes is settled.

Limit the number of hops / edges from the start node. Only find shortest-paths with a limited number of edges. We will call this *hop limit*.

背景

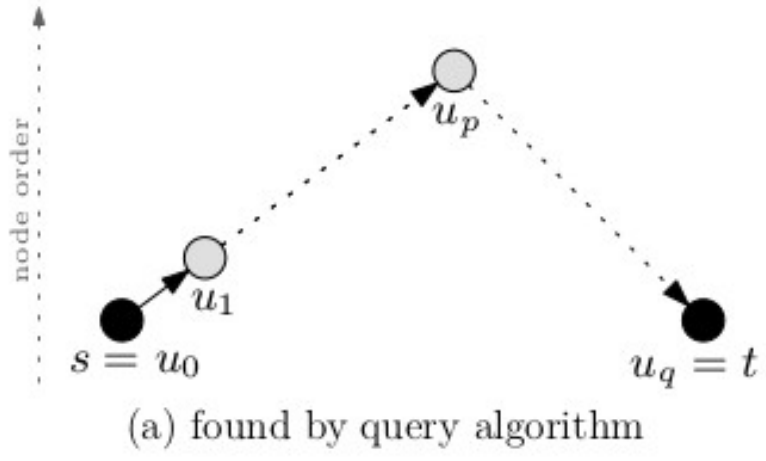
- Witness path search (确定 v->u->w 是不是 v 到 w 唯一的最短路)
  - 预处理需要找很多次 witness path ， Dijkstra 太慢怎么办？
    - 限制搜索范围的几种方法
      - CH 针对搜索深度为 1 和 2 的情况特别优化过了



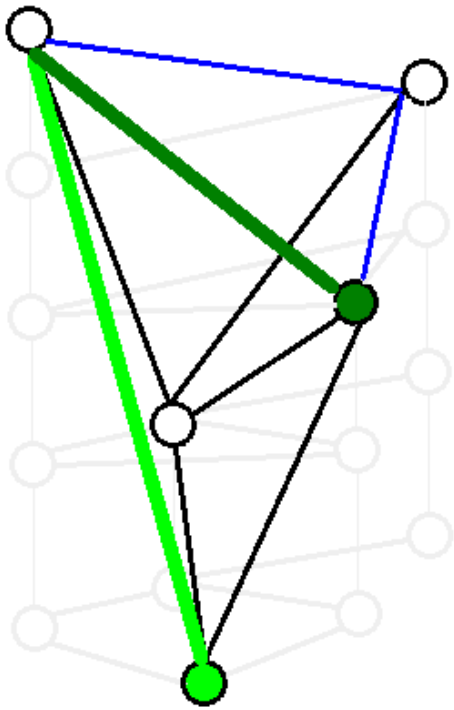


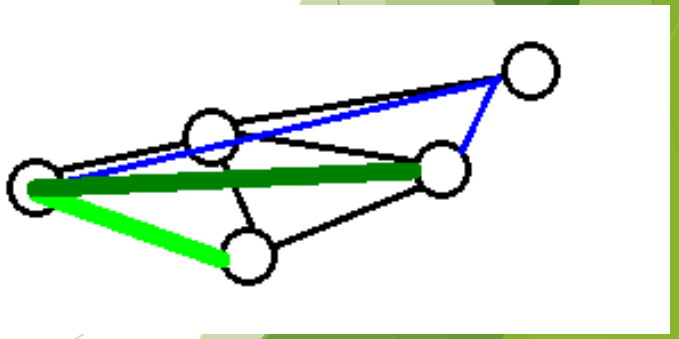
背景

- CH 查询最短路
  - 搜索时做双向 Dijkstra 搜索
  - 但有一个限制
    - 起点开始只往上层搜索，终点开始只往上层回溯



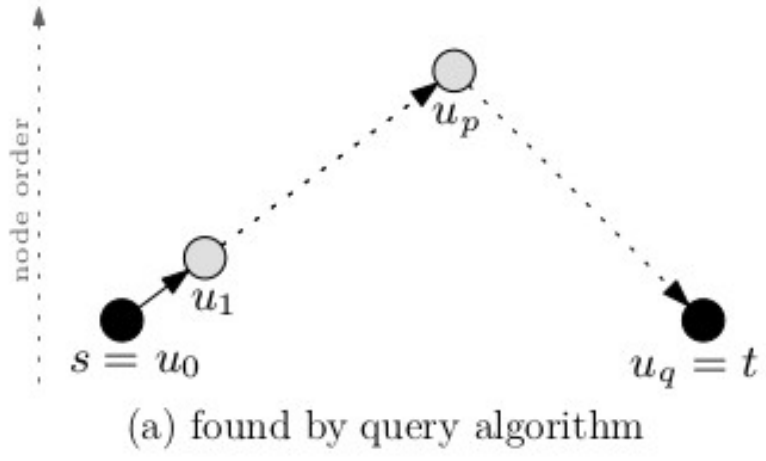
(a) found by query algorithm



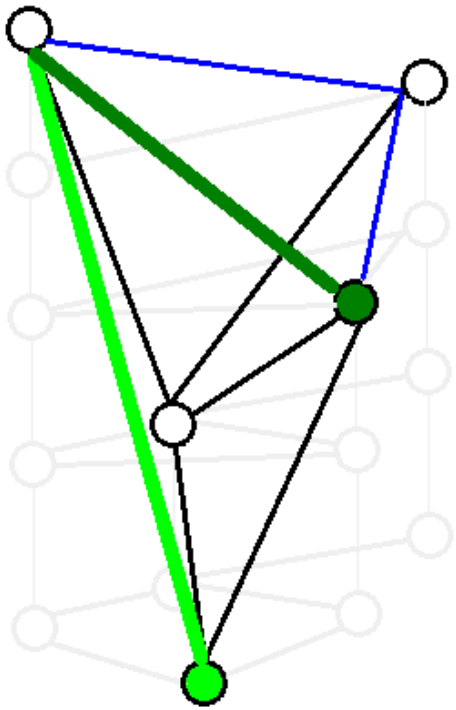


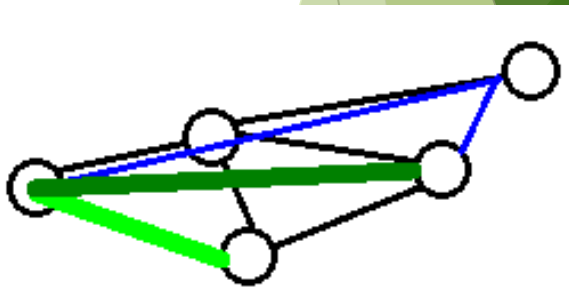
背景

- CH 查询最短路（优化？）
  - 搜索时做双向 Dijkstra 搜索
  - 但有一个限制
    - 起点开始只往上层搜索，终点开始只往上层回溯



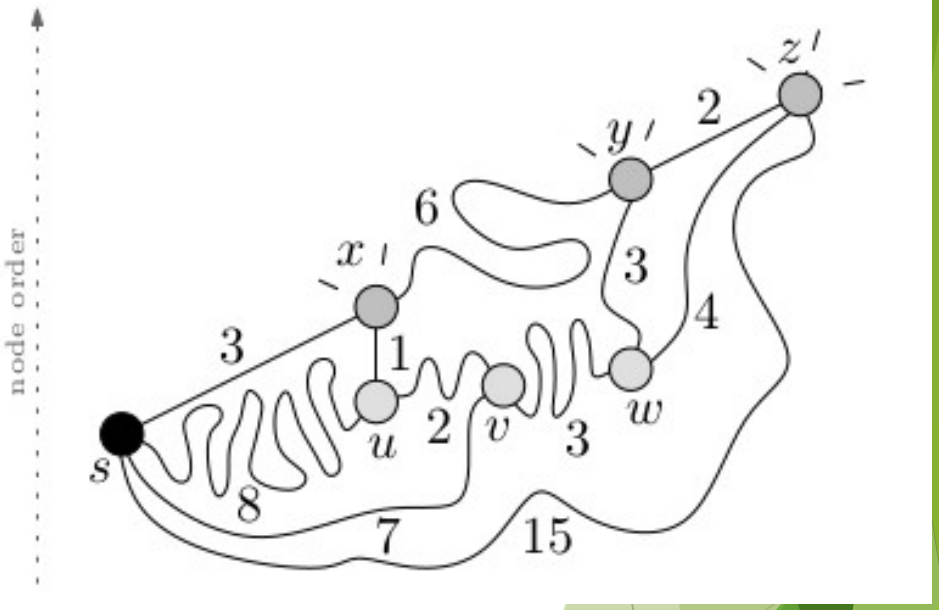
(a) found by query algorithm





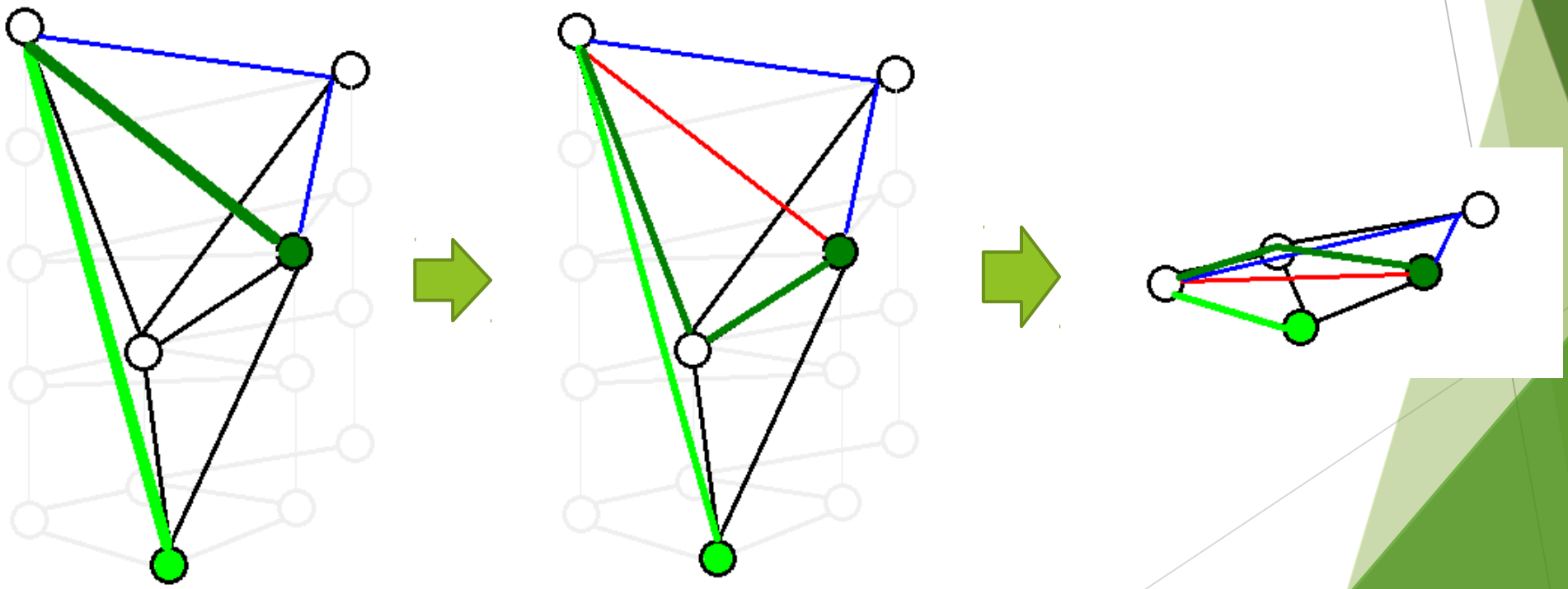
背景

- Stall-on-demand 优化
  - s->x->u 的路径不会被 CH 发现
  - s->u 的距离会是 8
  - 用 s->x->u 可以知道 s->u 肯定不是最短路
  - 所以可以 stall(阻止) 点 u 和其后继节点
  - 等到第二次被更新时解除 stall （策略不唯一）



背景

- CH 查询最短路
  - 得到最短路径后解压 shortcuts 得到对应原图的路径



CH 层次图构建

CH 层次图构建

- 代码 Overview (ch-compiler/src)

```
ch_adjgraph_types.h   ch_csrgraph_types.h     ch_dijkstra.h          ch_reader.h
ch_basetypes.h        ch_data_controller.cpp  ch_graph_builder.cpp   ch_utils.cpp
ch_bfs_alt.cpp        ch_data_controller.h    ch_graph_builder.h     ch_utils.h
ch_bfs_alt.h          ch_dataunit.cpp         ch_heap.hpp            console
ch_construct_ch.cpp   ch_dataunit.h           ch_map.hpp
ch_construct_ch.h     ch_dijkstra.cpp         ch_reader.cpp
```

CH 层次图构建

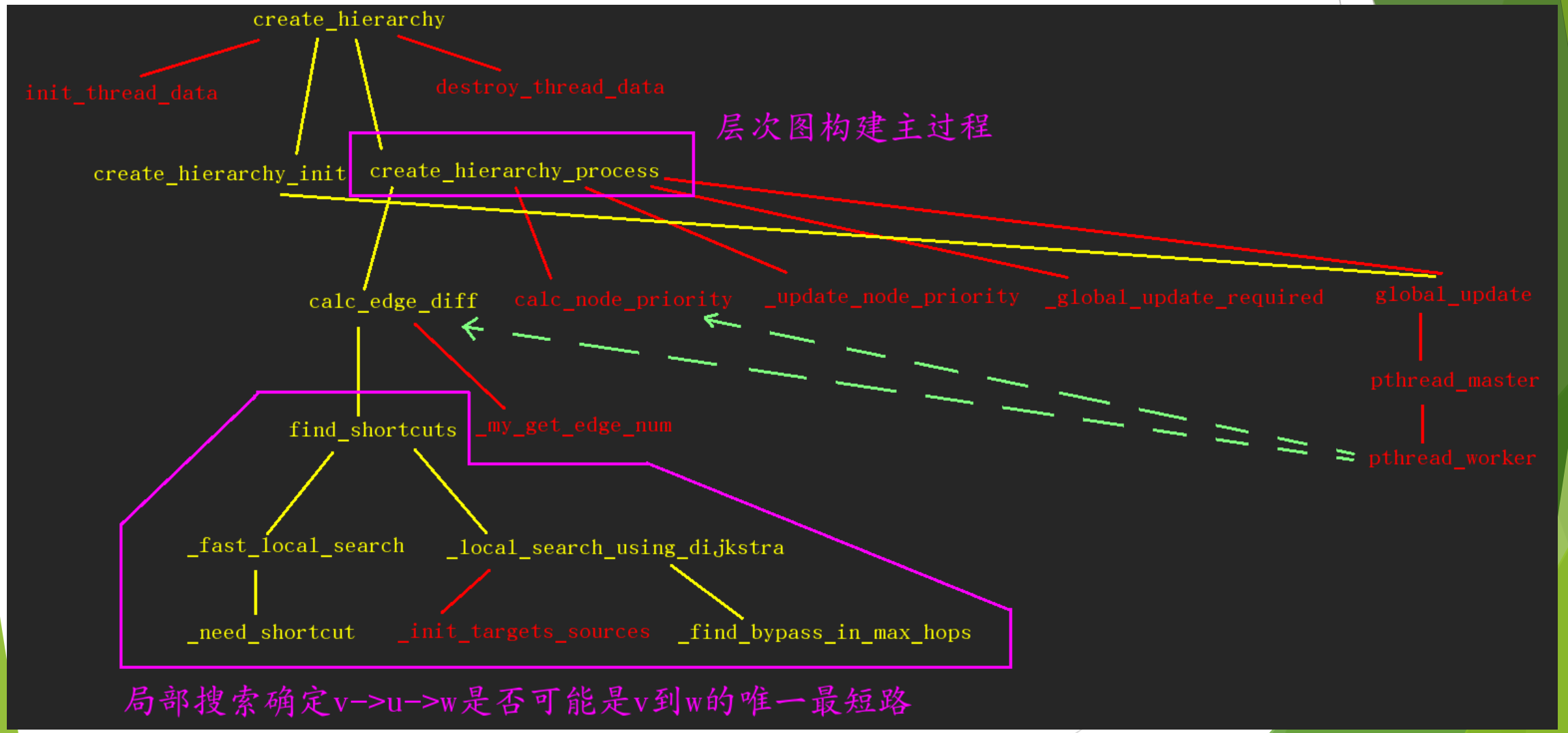
- 主要函数
  - src/ch_construct_ch.cpp（入口）
  - src/ch_utils.cpp （关键函数实现）

CH 层次图构建

- document 打开方式
  - 先读**黄字**和**橙字**，还有大号字体
  - 第二次读时再读所有注释
  - 最后一次结合代码读

CH 层次图构建

- 总流程


create_hierarchy
init_thread_data
destroy_thread_data
层次图构建主过程
create_hierarchy_init
create_hierarchy_process
calc_edge_diff
calc_node_priority
_update_node_priority
_global_update_required
global_update
pthread_master
pthread_worker
find_shortcuts
_my_get_edge_num
_fast_local_search
_local_search_using_dijkstra
_need_shortcut
_init_targets_sources
_find_bypass_in_max_hops
局部搜索确定v->u->w是否可能是v到w的唯一最短路

CH 层次图构建

- create_hierarchy (in 'src/ch_construct_ch.cpp' line 480)

```
480 ch_ret_t ConstructCH::create_hierarchy(CHGraph& graph,
481         std::vector<node_id_t>& node_order_vec,
482         std::vector<shortcut_edge_t>& shortcut_edge_vec)
483 {
484     CINFO_LOG( "ConstructCH::create_hierarchy starts");
485
486     int node_num = boost::num_vertices(graph);
487     Utils::init_thread_data(node_num,_contract_params.thread_num);
488
489     //split the function as two subroutines just for readablity consideration
490     this->_create_hierarchy_init(graph, shortcut_edge_vec);
491     this->_create_hierarchy_process(graph,
492             node_order_vec,
493             shortcut_edge_vec);
494
495     Utils::destroy_thread_data(_contract_params.thread_num);
496     CINFO_LOG( "ConstructCH::create_hierarchy ends");
497     return CH_RET_OK;
498 }
```

输入：一个图
输出：从前到后缩的点，添加的shortcut边
作用：根据给定图构建CH层次图

层次图构建过程中有并行操作，需要使用多线程，所以这里进行线程数据的初始化

各种数据初始化，主要是初始化了各个点的contract优先度

层次图构建主要过程

销毁线程数据

CH 层次图构建

- create_hierarchy_process (in 'src/ch_construct_ch.cpp' line 190)

```
189
190 ch_ret_t ConstructCH::_create_hierarchy_process(CHGraph& graph,
191         std::vector<node_id_t>& node_order_vec,
192         std::vector<shortcut_edge_t>& all_shortcut_vec)
193 {
194     int numVertex = boost::num_vertices(graph);
195     int numEdges = boost::num_edges(graph);
196     CINFO_LOG( "ConstructCH::create_hierarchy_process starts");
197
198     weight_map_t weight_map = boost::get(boost::edge_weight, graph);
199     shortcut_flag_map_t shortcut_flag_map = boost::get(boost::edge_shortcut_flag, graph);
200     node_order_map_t node_order_map = boost::get(boost::vertex_order, graph);
201     edge_id_map_t edge_id_map = boost::get(boost::edge_id, graph);
202
203     EdgeIDAllocator edge_id_allocator(numEdges);
204
205     int shortcuts_added = 0;
206     order_t current_node_order = 0; //record current contract node order
207
```

输入：图
输出：从前到后缩的点，添加的shortcut边
作用：构建CH层次图

边权“数组”（boost::property_map）
是不是shortcut“数组”
第几个被contract“数组”
每条边的id（注意shortcut的id需要新分配）

用来分配新的edge的id

```
209     //debug init;
210     // g_calc_edge_diff_time.reserve(boost::num_vertices(graph));
211     // g_find_shortcut_search_space.reserve(boost::num_vertices(graph));
212
213     for (size_t i = 0; i < boost::num_vertices(graph); ++i) {
214         node_id_t contract_node(0);
215         double myPriority = 0;
216         int reinsert_count = 0; /// debug purpose variable
217         std::vector<shortcut_info_t> shortcut_vec;
218         while (this->_node_pqueue_ptr->empty() != true) {
219             //handle to node
220             node_id_t min_node = this->_node_pqueue_ptr->topKey();
221             double minValue = _node_pqueue_ptr->topValue();
222
223             this->_node_pqueue_ptr->pop();
224
225             if (this->_node_pqueue_ptr->empty()) {
226                 contract_node = min_node;
227                 break;
228             }
229
230             int edge_diff = Utils::calc_edge_diff(
231                 shortcut_vec,
232                 (node_descriptor_t)min_node,
233                     graph,
234                     _node_deleted_flag_vec,
235                     _contract_params.max_hops);
236             _node_edge_diff_vec[min_node] = edge_diff;
237
238             pq_weight_t min_value = Utils::calc_node_priority(
239                     min_node,
240                     _node_edge_diff_vec[min_node],
241                     _node_deleted_neighbours_vec[min_node],
242                     _voronoi_number[min_node],
243                     _contract_params.term_coefficients);
244             myPriority = min_value;
245             node_id_t second_min_node = this->_node_pqueue_ptr->topKey();
246             pq_weight_t second_min_value = this->_node_pqueue_ptr->topValue();
247             if (min_value <= second_min_value) {
248                 contract_node = min_node;
249                 CDEBUG_LOG("reinsert_count:%d", reinsert_count);
250                 break;
251             }
252
253             this->_node_pqueue_ptr->push_or_update(min_node, min_value);
254             reinsert_count ++;
255         }
256
```

Contract主过程
缩n次，缩完为止

拿出当前优先度最小的点
（如果只有一个点，结束）

Lazy update
求最小优先度点

求一个点的
edge difference，
（顺便把shortcut算出来）
然后求其优先度

如果还是最小的，停止；
否则塞回去，重复

Contract主过程

```
257     if (i % _contract_params.debug_trace_cycle == 0) {
258         CINFO_LOG( "%u nodes are contracted", i);
259         CINFO_LOG( "%u shortcut are added", _shortcut_count);
260 
261         uint32_t vertex_num = numVertex;//boost::num_vertices(graph) - i;
262         uint32_t edge_num = numEdges;//boost::num_edges(graph) - _deleted_edge_count;
263         double graph_degree = edge_num * 1.0 / vertex_num;
264 
265         CINFO_LOG( "[nodes: %u] [edges: %u] [graph degree: %f]",
266                 vertex_num,
267                 edge_num,
268                 graph_degree);
269     }
270 
271     node_order_vec.push_back(contract_node);
272     node_order_map[contract_node] = current_node_order;
273     current_node_order ++;
274 
275     //std::vector<shortcut_edge_t> shortcut_vec;
276     //Utils::find_shortcuts(contract_node, graph, _node_deleted_flag_vec, 5, shortcut_vec);
277     shortcuts_added += shortcut_vec.size();
278     //add shortcut edges to graph edge
279     for (size_t i = 0; i < shortcut_vec.size(); ++i) {
280 
281         edge_descriptor_t first_edge = shortcut_vec[i].first_edge;
282         node_descriptor_t start_node = boost::source(first_edge, graph);
283 
284         edge_descriptor_t second_edge = shortcut_vec[i].second_edge;
285         node_descriptor_t end_node = boost::target(second_edge, graph);
286 
287         // set edge ids
288         shortcut_edge_t shortcut;
289         shortcut.first_edge_id = edge_id_map[first_edge];
290         shortcut.second_edge_id = edge_id_map[second_edge];
291         shortcut.id = edge_id_allocator.get_next_id();
292         all_shortcut_vec.push_back(shortcut);
293 
294         edge_descriptor_t shortcut_edge;
295         bool flag;
296         boost::tie(shortcut_edge, flag) = boost::add_edge(start_node, end_node, graph);
297         weight_map[shortcut_edge] = shortcut_vec[i].weight;
298         shortcut_flag_map[shortcut_edge] = true;
299         edge_id_map[shortcut_edge] = shortcut.id;
300 
301         CDEBUG_LOG("add shortcut [start_node:%u] [end_node:%u]", start_node, end_node);
302     }
303 
304     // record the shortcus
305 
306     // all_shortcut_vec.insert(all_shortcut_vec.end(),
307     //     shortcut_vec.begin(), shortcut_vec.end());
308 
309     numEdges += shortcut_vec.size();
310 
311     //remove in-edges and out-edges of node v from graph
312     // mark node as deleted
313     _node_deleted_flag_vec[contract_node] = true;
314 
315     _deleted_edge_count += _get_edge_num(contract_node, graph);
316     _shortcut_count += shortcut_vec.size();
317 
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320 
```

输出debug信息

更新contract node和shortcut相关的信息

比如shortcut是v->u->w

点v

点w

真正地把shortcut都加到图里面
(刚算出来的时候只是存在数组里)

设置一堆id属性：
v->u的id是啥
u->w的id是啥
v->w的id是啥

设权重/shortcut标记/新添shortcut v->w的id

总边数更新

contract掉的点对整个图的影响

这个点被contract掉了

删掉了这么多边
加上了这么多边

理论上是更新voronoi信息，实际上是什么都没做.....

```
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320
321     int goodEdge = 0;
322     //update neighbours priority
323     //adjacent vertices update
324     //com_writelog(COMLOG_DEBUG, "update neighbours priority starts");
325     std::pair<adjacency_iterator, adjacency_iterator> adj_vertices =
326         boost::adjacent_vertices(contract_node, graph);
327
328     for ( ;adj_vertices.first != adj_vertices.second; ++adj_vertices.first) {
329         node_id_t node_id = *(adj_vertices.first);
330
331         if (_node_deleted_flag_vec[node_id] == true) {
332             continue;
333         }
334         goodEdge ++;
335         this->_update_node_priority(node_id, graph);
336     }
337
338     //inv adjacent vertices update
339     std::pair<inv_adjacency_iterator, inv_adjacency_iterator> inv_adj_vertices =
340         boost::inv_adjacent_vertices(contract_node, graph);
341
342     for ( ;inv_adj_vertices.first != inv_adj_vertices.second; ++inv_adj_vertices.first) {
343         node_id_t node_id = *(inv_adj_vertices.first);
344
345         if (_node_deleted_flag_vec[node_id] == true) {
346             continue;
347         }
348         goodEdge ++;
349         this->_update_node_priority(node_id, graph);
350     }
351     //com_writelog(COMLOG_DEBUG, "update neighbours priority ends");
352
353     numEdges -= goodEdge;
354     numVertex --;
355     ///dump debug info
377     if (this->_global_update_required(numVertex, numEdges)) {
378         CINFO_LOG( "start global update");
379
380         // CINFO_LOG("before parallel\ndel_flag:%llx\ndel_nei:%llx\ne_diff:%llx,cp:%llx"
381         //注释&_node_deleted_flag_vec,
382         //     &_node_deleted_neighbours_vec,
383         //     &_node_edge_diff_vec,
384         //     &_contract_params);
385
386         Utils::global_update(graph,
387             _node_deleted_flag_vec,
388             _node_deleted_neighbours_vec,
389             _voronoi_number,
390             *_node_pqueue_ptr,
391             _node_edge_diff_vec,
392             _contract_params);
467         //注释
468         CINFO_LOG( "end global update");
469     }
470 }
471
472 /*if (com_log_enabled(COMLOG_DEBUG)) {
473     _debug_stream.close();
474 }*/
475 std::cout << "shortcuts_added=" << shortcuts_added << std::endl;
476 CINFO_LOG( "ConstructCH::create_hierarchy_process ends");
477 return CH_RET_OK;
478 }
```

Contract
主过程

Update
邻居节点

删边删点

全局Update

CH 层次图构建

- Utils::calc_edge_diff (in 'src/ch_utils.cpp' line 96)

```
96  int Utils::calc_edge_diff(std::vector<shortcut_info_t> & shortcut_vecs,
97      node_descriptor_t node,
98          const CHGraph& graph,
99          const std::vector<bool>& node_deleted_flag_vec,
100         uint32_t max_hops,
101         int pthread_id)
102 {
103     Utils::find_shortcuts(node,
104             //NOTE: we do no modification in the find_shortcuts function, const_cast
105             //is just for compile purpose
106             const_cast<CHGraph&>(graph),
107             node_deleted_flag_vec,
108             max_hops,
109             shortcut_vecs,
110             pthread_id);
111     // statistic_shortcut_num = shortcut_vecs.size();
112     // statistic_node_in_degree = boost::in_degree(node, graph);
113     // statistic_node_out_degree = boost::out_degree(node, graph);
114
115     /*
116     com_writelog(COMLOG_DEBUG, "node_id:%d", node);
117     com_writelog(COMLOG_DEBUG, "shortcut_vec size:%u", shortcut_vecs.size());
118     com_writelog(COMLOG_DEBUG, "node in degree:%d", boost::in_degree(node, graph));
119     com_writelog(COMLOG_DEBUG, "node out degree:%d", boost::out_degree(node, graph));
120     */
121     int edge_diff = shortcut_vecs.size()*10 - _my_get_edge_num(node, graph, node_deleted_flag_vec);
122         //- boost::in_degree(node, graph)
123         //- boost::out_degree(node, graph);
124
125     return edge_diff;
126 }
```

输入：
...，被contract的点，图，已经被删除的点，
局部搜索时的hop限制，并行时的pthread id
输出：
需要添加的shortcut
作用：
计算被contract点的edge difference，
顺便进行局部搜索把需要添加的shortcut算出来

进行witness path
局部搜索

计算edge difference，
此处为shortcut数*10-点度数

CH 层次图构建

- Utils::find_shortcuts (in 'src/ch_utils.cpp' line 504)

```
504 int Utils::find_shortcuts(node_descriptor_t vertex,
505         CHGraph& graph,
506         const std::vector<bool>& node_deleted_flag_vec,
507         int max_hops,
508         std::vector<shortcut_info_t>& shortcut_vecs,
509         int pthread_id)
510 {
511     if(true == node_deleted_flag_vec[node_descriptor2node_id(vertex)]) {
512         return CH_RET_ERROR;
513     }
514     shortcut_vecs.clear();
515
516     if(max_hops==1 || max_hops==2) {
517         CWARNING_LOG("using fast local search!");
518         return  fast_local_search(vertex,graph,node_deleted_flag_vec,max_hops,shortcut_vecs);
519     } else if (max_hops > 2) {
520         return _local_search_using_dijkstra(vertex,graph,node_deleted_flag_vec,
521             max_hops,shortcut_vecs,pthread_id);
522     } else {
523         return CH_RET_ERROR;
524     }
525 }
```

输入：被contract的点，图，已经被删除的点，
局部搜索的hop限制，...，并行时的pthread id

输出：添加的shortcut

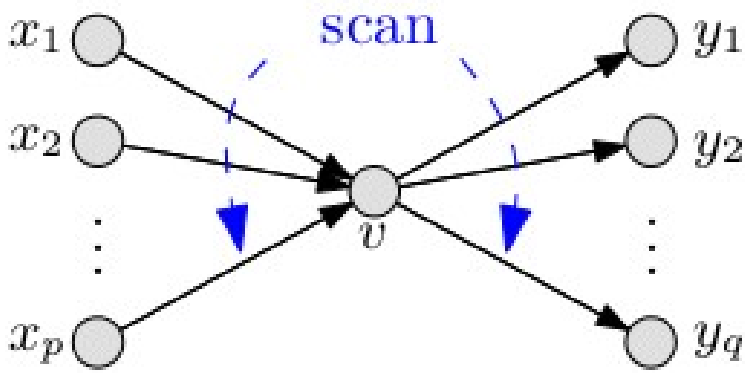
作用：进行局部witness path搜索，
得到需要添加的shortcuts

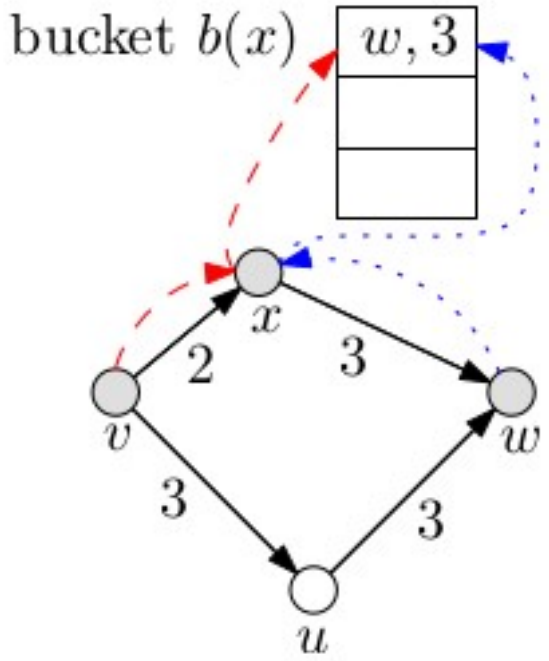
该点已经删除

hop为1或2时的
快速局部搜索

局部搜索

- Witness path search (确定 v->u->w 是不是 v 到 w 唯一的最短路)
  - 预处理需要找很多次 witness path ， Dijkstra 太慢怎么办？
    - 限制搜索范围的几种方法
      - CH 针对搜索深度为 1 和 2 的情况特别优化过了





CH 层次图构建

- Utils::_fast_local_search (in 'src/ch_utils.cpp' line 188)

CH 层次图构建

```
int Utils::_fast_local_search(node_descriptor_t vertex,
        CHGraph& graph,
        const std::vector<bool>& node_deleted_flag_vec,
        int max_hops,
        std::vector<shortcut_info_t>& shortcut_vecs)
{
    //for init static values
    static bool first = true;
    static std::vector<int> map_index;
    static std::vector<int> map_vis;
    static std::vector<weight_t> map_weight;

    if(first) {
        int node_num = boost::num_vertices(graph);
        //std::cout << "node_num = " << node_num << std::endl;
        map_index.reserve(node_num);
        map_vis.resize(node_num, false);
        map_weight.resize(node_num);
        first = false;
    }

    const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
    in_edge_iterator_t in_e, in_e_end;
    boost::tie(in_e, in_e_end) = boost::in_edges(vertex, graph);
    for (; in_e!=in_e_end; ++ in_e) {
        node_descriptor_t in_node = boost::source(*in_e, graph);
        weight_t in_edge_weight = edge_weight_map[*in_e];
        if ( in_edge_weight >= MAX_EDGE_WEIGHT ) { //ignore this edge.
            continue;
        }
        if(true == node_deleted_flag_vec[in_node]) {    //this node has been contracted
            continue;
        }
        if(in_node == vertex) {    //should not exist, but just in case.
            continue;
        }
        //clear map_index
        if(map_index.size() != 0) {
            for(size_t i = 0; i < map_index.size(); i ++) {//clear map_vis
                int index = map_index[i];
                map_vis[index] = false;
            }
            map_index.clear();
        }

        {//calculate map_weight
            out_edge_iterator_t in_out_e, in_out_e_end;
            boost::tie(in_out_e, in_out_e_end) = boost::out_edges(in_node, graph);
            for (; in_out_e!=in_out_e_end; ++in_out_e) {
                node_descriptor_t in_out_node = boost::target(*in_out_e, graph);
                weight_t in_out_edge_weight = edge_weight_map[*in_out_e];
                if (in_out_edge_weight >= MAX_EDGE_WEIGHT ) { //ignore this edge.
                    continue;
                }
                if(true == node_deleted_flag_vec[in_out_node]) {    //this node has been contracted
                    continue;
                }
                if(in_out_node == vertex) {
                    continue;
                }
                if(false == map_vis[in_out_node]) {
                    map_vis[in_out_node] = true;
                    map_weight[in_out_node] = in_out_edge_weight;
                    map_index.push_back(in_out_node);
                } else {
                    map_weight[in_out_node] = std::min(map_weight[in_out_node], in_out_edge_weight);
                }
            }
        }
```

输入：被contract的点，图，
已经被删除的点，局部搜索时的hop限制
输出：需要添加的shortcuts
作用：hop限制为1或2的局部搜索

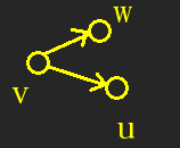
第一次调用
初始化数据结构
map_index: 1-hop能到的节点集
map_vis: 一个点是否在map_index中
map_weight: 1-hop到该点的最短路径长度

3条件：
合法边
未被删
非自环

动态清空
数据结构

枚举'v'->u

枚举v->'w'



更新1-hop
path的信息

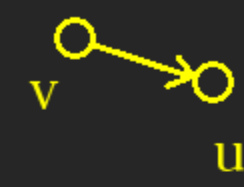
CH 层次图构建

```
258         out_edge_iterator_t out_e, out_e_end;
259         boost::tie(out_e, out_e_end) = boost::out_edges(vertex, graph);
260
261         //check and add shortcut
262         for (; out_e!=out_e_end; ++out_e) {
263             node_descriptor_t out_node = boost::target(*out_e, graph);
264             if (out_node == in_node) {
265                 continue;
266             }
267             weight_t out_edge_weight = edge_weight_map[*out_e];
268             if (out_edge_weight >= MAX_EDGE_WEIGHT) {
269                 continue;
270             }
271             weight_t shortcut_weight = out_edge_weight  + in_edge_weight;
272             if (true == _need_shortcut(vertex,out_node,shortcut_weight,graph,node_deleted_flag_vec,
273                 max_hops,map_vis,map_weight)) {
274                 shortcut_info_t shortcut;
275                 shortcut.first_edge = *in_e;
276                 shortcut.second_edge = *out_e;
277                 shortcut.weight = shortcut_weight;
278
279                 shortcut_vecs.push_back(shortcut);
280             }
281         }
282     }
283     return CH_RET_OK;
284 }
```

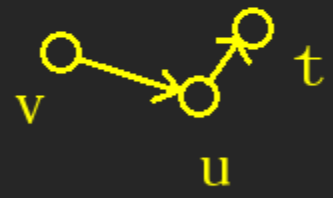
利用当前信息枚举v->t之间的所有1或2hop路径

添加shortcuts

枚举'v'->u



枚举u->'t'



CH 层次图构建

- Utils::_need_shortcut (in 'src/ch_utils.cpp' line 141)

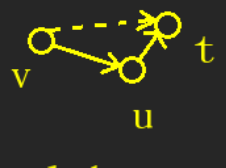
```
*/
bool _need_shortcut(node_descriptor_t vertex,
    node_descriptor_t out_node,
    weight_t shortcut_weight,
    CHGraph& graph,
    const std::vector<bool>& node_deleted_flag_vec,
    int max_hops,
    const std::vector<int>& map_vis,
    const std::vector<weight_t>& map_weight)
{
    const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
    if(true == node_deleted_flag_vec[out_node]) {
        return false;
    }
    if(out_node == vertex) {    //should not exist, but just in case.
        return false;
    }

    if(true == map_vis[out_node]) {
        if(map_weight[out_node] <= shortcut_weight) {
            return false;
        }
    }
    if(2 == max_hops) {
        in_edge_iterator_t out_in_e, out_in_e_end;
        boost::tie(out_in_e, out_in_e_end) = boost::in_edges(out_node, graph);
        for (; out_in_e!=out_in_e_end; ++ out_in_e) {
            node_descriptor_t out_in_node = boost::source(*out_in_e, graph);
            weight_t out_in_edge_weight = edge_weight_map[*out_in_e];
            if (out_in_edge_weight >= MAX_EDGE_WEIGHT) { //ignore this edge.
                continue;
            }
            if(true == node_deleted_flag_vec[out_in_node]) {    //thi  been contracted
                continue;
            }
            if(out_in_node == vertex) {
                continue;
            }
            if(true == map_vis[out_in_node]) {
                if(map_weight[out_in_node]+out_in_edge_weight<=shortcut_weight) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

输入：被contract的点u，目标点t，
v->u->t的路径长度，图，
已经被删除的点，局部搜索的hop限制，
点w是否在v->w上，
点w如果在v->w上v->w的长度
作用：是否存在v->t的比v->u->t更短的1/2-hop路径

未被删
非自环

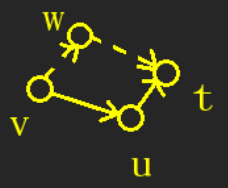
找到1-hop
的更短路径



1-hop

枚举'w'->t

3条件：
合法边
未被删
非自环



2-hop

比较v->w->t的长度
和v->u->t的长度
如果更短，则
找到2-hop的更
短路径

CH 层次图构建

- Utils::_fast_local_search (in 'src/ch_utils.cpp' line 188)

CH 层次图构建

```
int Utils::_fast_local_search(node_descriptor_t vertex,
        CHGraph& graph,
        const std::vector<bool>& node_deleted_flag_vec,
        int max_hops,
        std::vector<shortcut_info_t>& shortcut_vecs)
{
    //for init static values
    static bool first = true;
    static std::vector<int> map_index;
    static std::vector<int> map_vis;
    static std::vector<weight_t> map_weight;

    if(first) {
        int node_num = boost::num_vertices(graph);
        //std::cout << "node_num = " << node_num << std::endl;
        map_index.reserve(node_num);
        map_vis.resize(node_num, false);
        map_weight.resize(node_num);
        first = false;
    }

    const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
    in_edge_iterator_t in_e, in_e_end;
    boost::tie(in_e, in_e_end) = boost::in_edges(vertex, graph);
    for (; in_e!=in_e_end; ++ in_e) {
        node_descriptor_t in_node = boost::source(*in_e, graph);
        weight_t in_edge_weight = edge_weight_map[*in_e];
        if ( in_edge_weight >= MAX_EDGE_WEIGHT ) { //ignore this edge.
            continue;
        }
        if(true == node_deleted_flag_vec[in_node]) {    //this node has been contracted
            continue;
        }
        if(in_node == vertex) {    //should not exist, but just in case.
            continue;
        }
        //clear map_index
        if(map_index.size() != 0) {
            for(size_t i = 0; i < map_index.size(); i ++) {//clear map_vis
                int index = map_index[i];
                map_vis[index] = false;
            }
            map_index.clear();
        }

        {//calculate map_weight
            out_edge_iterator_t in_out_e, in_out_e_end;
            boost::tie(in_out_e, in_out_e_end) = boost::out_edges(in_node, graph);
            for (; in_out_e!=in_out_e_end; ++in_out_e) {
                node_descriptor_t in_out_node = boost::target(*in_out_e, graph);
                weight_t in_out_edge_weight = edge_weight_map[*in_out_e];
                if (in_out_edge_weight >= MAX_EDGE_WEIGHT ) { //ignore this edge.
                    continue;
                }
                if(true == node_deleted_flag_vec[in_out_node]) {    //this node has been contracted
                    continue;
                }
                if(in_out_node == vertex) {
                    continue;
                }
                if(false == map_vis[in_out_node]) {
                    map_vis[in_out_node] = true;
                    map_weight[in_out_node] = in_out_edge_weight;
                    map_index.push_back(in_out_node);
                } else {
                    map_weight[in_out_node] = std::min(map_weight[in_out_node], in_out_edge_weight);
                }
            }
        }
```

输入：被contract的点，图，
已经被删除的点，局部搜索时的hop限制
输出：需要添加的shortcuts
作用：hop限制为1或2的局部搜索

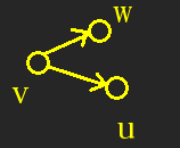
第一次调用
初始化数据结构
map_index: 1-hop能到的节点集
map_vis: 一个点是否在map_index中
map_weight: 1-hop到该点的最短路径长度

3条件：
合法边
未被删
非自环

动态清空
数据结构

枚举'v'->u

枚举v->'w'



更新1-hop
path的信息

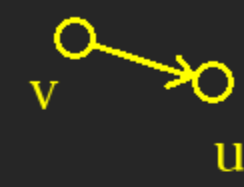
CH 层次图构建

```
258         out_edge_iterator_t out_e, out_e_end;
259         boost::tie(out_e, out_e_end) = boost::out_edges(vertex, graph);
260
261         //check and add shortcut
262         for (; out_e!=out_e_end; ++out_e) {
263             node_descriptor_t out_node = boost::target(*out_e, graph);
264             if (out_node == in_node) {
265                 continue;
266             }
267             weight_t out_edge_weight = edge_weight_map[*out_e];
268             if (out_edge_weight >= MAX_EDGE_WEIGHT) {
269                 continue;
270             }
271             weight_t shortcut_weight = out_edge_weight  + in_edge_weight;
272             if (true == _need_shortcut(vertex,out_node,shortcut_weight,graph,node_deleted_flag_vec,
273                 max_hops,map_vis,map_weight)) {
274                 shortcut_info_t shortcut;
275                 shortcut.first_edge = *in_e;
276                 shortcut.second_edge = *out_e;
277                 shortcut.weight = shortcut_weight;
278
279                 shortcut_vecs.push_back(shortcut);
280             }
281         }
282     }
283     return CH_RET_OK;
284 }
```

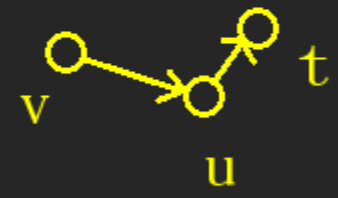
利用当前信息枚举v->t之间的所有1或2hop路径

添加shortcuts

枚举'v'->u



枚举u->'t'



CH 层次图构建

- Utils::find_shortcuts (in 'src/ch_utils.cpp' line 504)

```
504 int Utils::find_shortcuts(node_descriptor_t vertex,
505         CHGraph& graph,
506         const std::vector<bool>& node_deleted_flag_vec,
507         int max_hops,
508         std::vector<shortcut_info_t>& shortcut_vecs,
509         int pthread_id)
510 {
511     if(true == node_deleted_flag_vec[node_descriptor2node_id(vertex)]) {
512         return CH_RET_ERROR;
513     }
514     shortcut_vecs.clear();
515
516     if(max_hops==1 || max_hops==2) {
517         CWARNING_LOG("using fast local search!");
518         return  fast_local_search(vertex,graph,node_deleted_flag_vec,max_hops,shortcut_vecs);
519     } else if (max_hops > 2) {
520         return _local_search_using_dijkstra(vertex,graph,node_deleted_flag_vec,
521             max_hops,shortcut_vecs,pthread_id);
522     } else {
523         return CH_RET_ERROR;
524     }
525 }
```

输入：被contract的点，图，已经被删除的点，
局部搜索的hop限制，...，并行时的pthread id

输出：添加的shortcut

作用：进行局部witness path搜索，
得到需要添加的shortcuts

该点已经删除

hop为1或2时的
快速局部搜索

局部搜索

CH 层次图构建

- Utils::_local_search_using_dijkstra (in 'src/ch_utils.cpp' line 461)

```
461 int Utils::_local_search_using_dijkstra(node_descriptor_t vertex,
462         CHGraph& graph,
463         const std::vector<bool>& node_deleted_flag_vec,
464         int max_hops,
465         std::vector<shortcut_info_t>& shortcut_vecs,
466         int pthread_id)
467 {
468     WeightHopHeap &myHeap = *Utils::pthread_heap_ptr[pthread_id];
469     ch::map<int> &closeTable = *Utils::pthread_closeTable_ptr[pthread_id];
470     ch::map<edge_descriptor_t> &sources = *Utils::pthread_sources_ptr[pthread_id];
471     ch::map<std::pair<edge_descriptor_t,int> > &targets = *Utils::pthread_targets_ptr[pthread_id];
472
473     const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
474     weight_t max_out_edge_weight = 0 ;
475     _init_targets_sources(vertex, graph, node_deleted_flag_vec, edge_weight_map,
476         sources, targets, max_out_edge_weight);
477
478
479     for(uint32_t i_source = 0; i_source<sources.size(); i_source ++) {
480         node_descriptor_t in_node = sources.key(i_source);
481         edge_descriptor_t in_edge = sources[in_node];
482
483         weight_t in_edge_weight = edge_weight_map[in_edge];
484
485         _find_bypass_in_max_hops(vertex,in_node,in_edge_weight, in_edge_weight + max_out_edge_weight,
486             targets, max_hops, graph,
487             node_deleted_flag_vec, closeTable, myHeap, edge_weight_map);
488
489         for(uint32_t i_target = 0; i_target<targets.size(); i_target ++) {
490             node_descriptor_t out_node = targets.key(i_target);
491             if(targets[out_node].second != 1) {//need shortcuts
492                 shortcut_info_t shortcut;
493                 shortcut.first_edge = in_edge;
494                 shortcut.second edge = targets[out node].first;
495                 shortcut.weight = in_edge_weight + edge_weight_map[shortcut.second_edge];
496
497                 shortcut_vecs.push_back(shortcut);
498             }
499         }
500         // statistic_search_space_size += closeTable.size() + myHeap.size();
501     }
502     return CH_RET_OK;
503 }
```

输入：被contract的点，图，
已经被删除的点，局部搜索的hop限制，
并行时的pthread id
输出：需要添加的shortcut
作用：进行局部搜索，得到必须添加的shortcuts

dijkstra使用的优先队列
（记录<路径长度,路径hop>）
dijkstra过程中固定的点
局部搜索的所有起点'v'->u
局部搜索的所有终点u->'w'

初始化sources和targets，
得到所有起点和终点

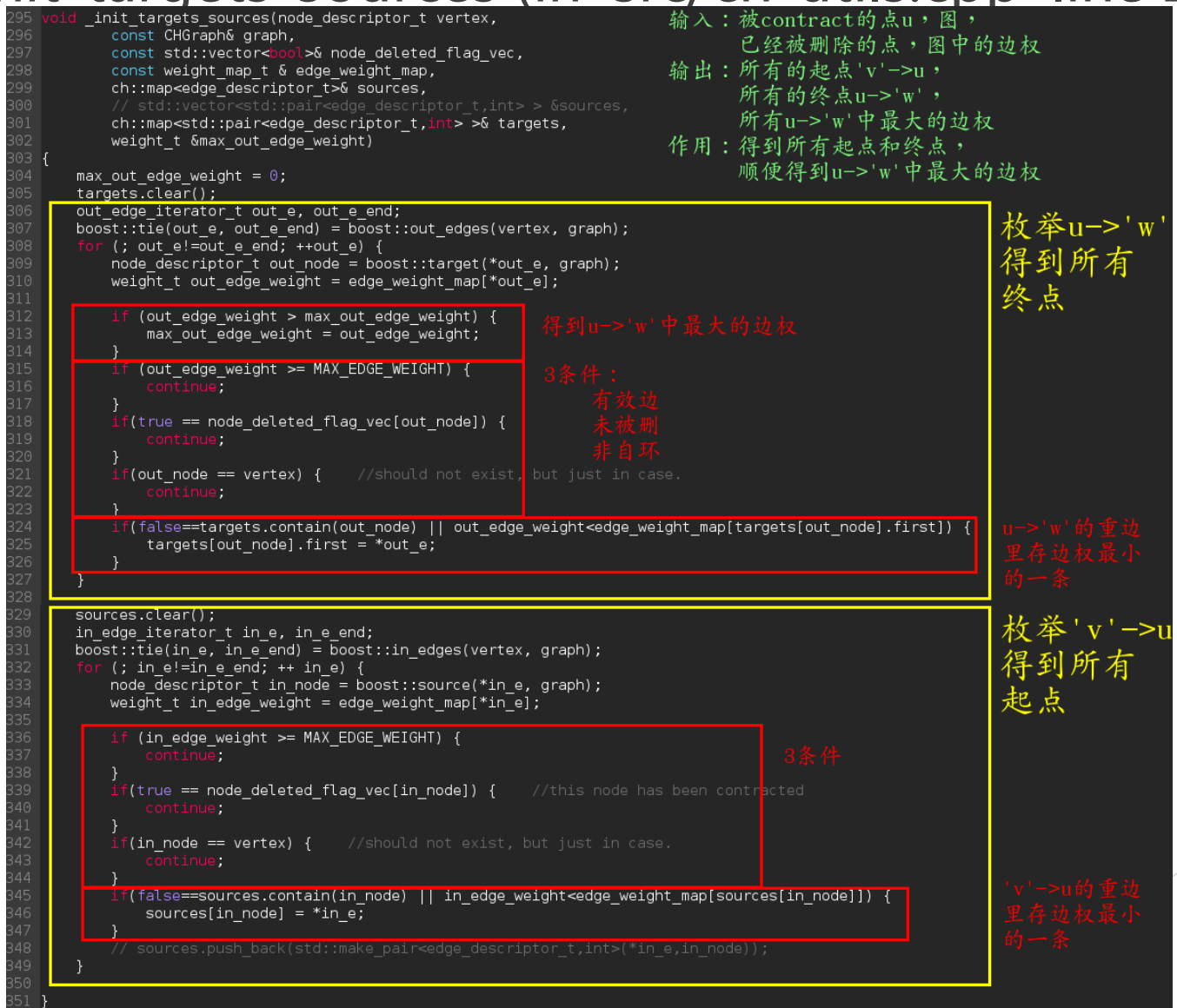
枚举所有起点
'v'->u

进行一对多的
局部搜索

枚举所有终点，
如果经过局部搜索
需要添加shortcut，
则添加
（targets数据结构：
first：边u->'w'
second：是否需要
添加shortcut）

CH 层次图构建

- Utils::_init_targets_sources (in 'src/ch_utils.cpp' line 295)



CH 层次图构建

- Utils::_local_search_using_dijkstra (in 'src/ch_utils.cpp' line 461)

```
461 int Utils::_local_search_using_dijkstra(node_descriptor_t vertex,
462         CHGraph& graph,
463         const std::vector<bool>& node_deleted_flag_vec,
464         int max_hops,
465         std::vector<shortcut_info_t>& shortcut_vecs,
466         int pthread_id)
467 {
468     WeightHopHeap &myHeap = *Utils::pthread_heap_ptr[pthread_id];
469     ch::map<int> &closeTable = *Utils::pthread_closeTable_ptr[pthread_id];
470     ch::map<edge_descriptor_t> &sources = *Utils::pthread_sources_ptr[pthread_id];
471     ch::map<std::pair<edge_descriptor_t,int> > &targets = *Utils::pthread_targets_ptr[pthread_id];
472
473     const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
474     weight_t max_out_edge_weight = 0 ;
475     _init_targets_sources(vertex, graph, node_deleted_flag_vec, edge_weight_map,
476         sources, targets, max_out_edge_weight);
477
478
479     for(uint32_t i_source = 0; i_source<sources.size(); i_source ++) {
480         node_descriptor_t in_node = sources.key(i_source);
481         edge_descriptor_t in_edge = sources[in_node];
482
483         weight_t in_edge_weight = edge_weight_map[in_edge];
484
485         _find_bypass_in_max_hops(vertex,in_node,in_edge_weight, in_edge_weight + max_out_edge_weight,
486             targets, max_hops, graph,
487             node_deleted_flag_vec, closeTable, myHeap, edge_weight_map);
488
489         for(uint32_t i_target = 0; i_target<targets.size(); i_target ++) {
490             node_descriptor_t out_node = targets.key(i_target);
491             if(targets[out_node].second != 1) {//need shortcuts
492                 shortcut_info_t shortcut;
493                 shortcut.first_edge = in_edge;
494                 shortcut.second edge = targets[out node].first;
495                 shortcut.weight = in_edge_weight + edge_weight_map[shortcut.second_edge];
496
497                 shortcut_vecs.push_back(shortcut);
498             }
499         }
500         // statistic_search_space_size += closeTable.size() + myHeap.size();
501     }
502     return CH_RET_OK;
503 }
```

输入：被contract的点，图，
已经被删除的点，局部搜索的hop限制，
并行时的pthread id
输出：需要添加的shortcut
作用：进行局部搜索，得到必须添加的shortcuts

dijkstra使用的优先队列
（记录<路径长度,路径hop>）
dijkstra过程中固定的点
局部搜索的所有起点'v'->u
局部搜索的所有终点u->'w'

初始化sources和targets，
得到所有起点和终点

枚举所有起点
'v'->u

进行一对多的
局部搜索

枚举所有终点，
如果经过局部搜索
需要添加shortcut，
则添加
（targets数据结构：
first：边u->'w'
second：是否需要
添加shortcut）

CH 层次图构建

- Utils::_find_bypass_in_max_hops (in 'src/ch_utils.cpp' line 364)

```
363  */
364 void _find_bypass_in_max_hops(node_descriptor_t vertex,
365     node_descriptor_t in_node,
366     const weight_t in_edge_weight,
367     const weight_t max_shortcut_length,
368     ch::map<std::pair<edge_descriptor_t,int> >& targets,
369     int max_hops,
370     CHGraph& graph,
371     const std::vector<bool>& node_deleted_flag_vec,
372     ch::map<int>& closeTable,
373     WeightHopHeap& myHeap,
374     const weight_map_t & edge_weight_map)
375 {
376
377     for(size_t i_target = 0; i_target<targets.size(); i_target ++) {
378         node_descriptor_t out_node = targets.key(i_target);
379         targets[out_node].second = 0;
380     }
381     closeTable.clear();
382     int count = targets.size();
383     myHeap.clear();
384     myHeap.push_or_update(in_node, WeightHopPair(0,0));
385
```

输入：被contract的点u，起点v，
v->u的长度，v->u->'w'的最大长度，
所有终点w，局部搜索的hop限制，
图，已经被删除的点，
dijkstra过程中被固定的点，
dijkstra过程中使用的优先队列，
图中的边权

输出：所有终点targets的second域
（v->u->w是否需要添加shortcut）

作用：进行一对多(v->...->'w')的局部搜索，
决定每对v->w是否需要添加shortcut

初始化targets的second域

初始化优先队列，只有一个点v（起点）

（key：节点
value：
first：路径长度
second：路径hop数）

CH 层次图构建

```
while(!myHeap.empty()) {
    int topKey = myHeap.topKey();
    WeightHopPair topVal = myHeap.topValue();

    // if (Utils::stat_max_open_table_size < myHeap.size()) {
    //     Utils::stat_max_open_table_size = myHeap.size();
    // }

    myHeap.pop();

    closeTable[topKey];

    if( true==targets.contain(topKey) ) {
        if(targets[topKey].second == 0) {

            if(topVal.first<=edge_weight_map[targets[topKey].first]+in_edge_weight) {
                targets[topKey].second = 1;
            } else {
                targets[topKey].second = 2;
            }
            count --;
            if(count == 0) {
                break;
            }
        }
    }
    if(topVal.second < max_hops) {
        out_edge_iterator_t e, e_end;
        boost::tie(e, e_end) = boost::out_edges(topKey, graph);
        for (; e!=e_end; ++ e) {
            node_descriptor_t node = boost::target(*e, graph);
            if(true == node_deleted_flag_vec[node]) {
                continue;
            }
            if(node == vertex) {
                continue;
            }
            if(closeTable.contain(node)) {
                continue;
            }
            weight_t edge_weight = edge_weight_map[*e];
            if(edge_weight >= MAX_EDGE_WEIGHT) {
                continue;
            }

            weight_t dist = topVal.first + edge_weight;

            if (dist > max_shortcut_length) {
                continue;
            }
            if(false==myHeap.contain(node) || dist<myHeap[node].first) {
                myHeap.push_or_update(node, WeightHopPair(dist, topVal.second+1));
            }

            if(true==targets.contain(node)) {
                if(targets[node].second == 0) {
                    if(dist<=edge_weight_map[targets[node].first]+in_edge_weight) {
                        targets[node].second = 1;
                        count --;
                        if(count == 0) {
                            break;
                        }
                    }
                }
            }
        }
    }
    if(count == 0) {
        break;
    }
}
// if (Utils::stat_max_close_table_size < closeTable.size()) {
//     Utils::stat_max_close_table_size = closeTable.size();
// }
}
```

取优先队列队头进行扩展

从队列中弹出该点
固定该点

3条件
- 未被删
- 非自环
- 有效边

+没有被固定

如果当前路径长度
超过任何一对
v->u->'w'的长度，
则其不可能是
witness path

一对多
Dijkstra
局部搜索

到达某个终点w，
判断路径长度是否比
v->u->w的长度更短

由于该点已经被固定，
所以不会有更短的路径了，
v->...->w的最短路已经找到

如果所有终点的最短路都
找到，则可以退出

hop小于限制
扩展得到新节点

加入优先队列
或更新优先队列中的该点

扩展后到达某个终点w，
而且长度比v->u->w短，
则找到v->u->w的一条
witness path

CH 层次图构建

- Utils::_local_search_using_dijkstra (in 'src/ch_utils.cpp' line 461)

```
461 int Utils::_local_search_using_dijkstra(node_descriptor_t vertex,
462         CHGraph& graph,
463         const std::vector<bool>& node_deleted_flag_vec,
464         int max_hops,
465         std::vector<shortcut_info_t>& shortcut_vecs,
466         int pthread_id)
467 {
468     WeightHopHeap &myHeap = *Utils::pthread_heap_ptr[pthread_id];
469     ch::map<int> &closeTable = *Utils::pthread_closeTable_ptr[pthread_id];
470     ch::map<edge_descriptor_t> &sources = *Utils::pthread_sources_ptr[pthread_id];
471     ch::map<std::pair<edge_descriptor_t,int> > &targets = *Utils::pthread_targets_ptr[pthread_id];
472
473     const weight_map_t & edge_weight_map = boost::get(boost::edge_weight, graph);
474     weight_t max_out_edge_weight = 0 ;
475     _init_targets_sources(vertex, graph, node_deleted_flag_vec, edge_weight_map,
476         sources, targets, max_out_edge_weight);
477
478
479     for(uint32_t i_source = 0; i_source<sources.size(); i_source ++) {
480         node_descriptor_t in_node = sources.key(i_source);
481         edge_descriptor_t in_edge = sources[in_node];
482
483         weight_t in_edge_weight = edge_weight_map[in_edge];
484
485         _find_bypass_in_max_hops(vertex,in_node,in_edge_weight, in_edge_weight + max_out_edge_weight,
486             targets, max_hops, graph,
487             node_deleted_flag_vec, closeTable, myHeap, edge_weight_map);
488
489         for(uint32_t i_target = 0; i_target<targets.size(); i_target ++) {
490             node_descriptor_t out_node = targets.key(i_target);
491             if(targets[out_node].second != 1) {//need shortcuts
492                 shortcut_info_t shortcut;
493                 shortcut.first_edge = in_edge;
494                 shortcut.second edge = targets[out node].first;
495                 shortcut.weight = in_edge_weight + edge_weight_map[shortcut.second_edge];
496
497                 shortcut_vecs.push_back(shortcut);
498             }
499         }
500         // statistic_search_space_size += closeTable.size() + myHeap.size();
501     }
502     return CH_RET_OK;
503 }
```

输入：被contract的点，图，
已经被删除的点，局部搜索的hop限制，
并行时的pthread id
输出：需要添加的shortcut
作用：进行局部搜索，得到必须添加的shortcuts

dijkstra使用的优先队列
（记录<路径长度,路径hop>）
dijkstra过程中固定的点
局部搜索的所有起点'v'->u
局部搜索的所有终点u->'w'

初始化sources和targets，
得到所有起点和终点

枚举所有起点
'v'->u

进行一对多的
局部搜索

枚举所有终点，
如果经过局部搜索
需要添加shortcut，
则添加
（targets数据结构：
first：边u->'w'
second：是否需要
添加shortcut）

CH 层次图构建

- Utils::find_shortcuts (in 'src/ch_utils.cpp' line 504)

```
504 int Utils::find_shortcuts(node_descriptor_t vertex,
505         CHGraph& graph,
506         const std::vector<bool>& node_deleted_flag_vec,
507         int max_hops,
508         std::vector<shortcut_info_t>& shortcut_vecs,
509         int pthread_id)
510 {
511     if(true == node_deleted_flag_vec[node_descriptor2node_id(vertex)]) {
512         return CH_RET_ERROR;
513     }
514     shortcut_vecs.clear();
515
516     if(max_hops==1 || max_hops==2) {
517         CWARNING_LOG("using fast local search!");
518         return  fast_local_search(vertex,graph,node_deleted_flag_vec,max_hops,shortcut_vecs);
519     } else if (max_hops > 2) {
520         return _local_search_using_dijkstra(vertex,graph,node_deleted_flag_vec,
521             max_hops,shortcut_vecs,pthread_id);
522     } else {
523         return CH_RET_ERROR;
524     }
525 }
```

输入：被contract的点，图，已经被删除的点，
局部搜索的hop限制，...，并行时的pthread id

输出：添加的shortcut

作用：进行局部witness path搜索，
得到需要添加的shortcuts

该点已经删除

hop为1或2时的
快速局部搜索

局部搜索

CH 层次图构建

- Utils::calc_edge_diff (in 'src/ch_utils.cpp' line 96)

```
96  int Utils::calc_edge_diff(std::vector<shortcut_info_t> & shortcut_vecs,
97      node_descriptor_t node,
98          const CHGraph& graph,
99          const std::vector<bool>& node_deleted_flag_vec,
100         uint32_t max_hops,
101         int pthread_id)
102 {
103     Utils::find_shortcuts(node,
104             //NOTE: we do no modification in the find_shortcuts function, const_cast
105             //is just for compile purpose
106             const_cast<CHGraph&>(graph),
107             node_deleted_flag_vec,
108             max_hops,
109             shortcut_vecs,
110             pthread_id);
111     // statistic_shortcut_num = shortcut_vecs.size();
112     // statistic_node_in_degree = boost::in_degree(node, graph);
113     // statistic_node_out_degree = boost::out_degree(node, graph);
114
115     /*
116     com_writelog(COMLOG_DEBUG, "node_id:%d", node);
117     com_writelog(COMLOG_DEBUG, "shortcut_vec size:%u", shortcut_vecs.size());
118     com_writelog(COMLOG_DEBUG, "node in degree:%d", boost::in_degree(node, graph));
119     com_writelog(COMLOG_DEBUG, "node out degree:%d", boost::out_degree(node, graph));
120     */
121     int edge_diff = shortcut_vecs.size()*10 - _my_get_edge_num(node, graph, node_deleted_flag_vec);
122         //- boost::in_degree(node, graph)
123         //- boost::out_degree(node, graph);
124
125     return edge_diff;
126 }
```

输入：
...，被contract的点，图，已经被删除的点，
局部搜索时的hop限制，并行时的pthread id
输出：
需要添加的shortcut
作用：
计算被contract点的edge difference，
顺便进行局部搜索把需要添加的shortcut算出来

进行witness path
局部搜索

计算edge difference，
此处为shortcut数*10-点度数

CH 层次图构建

- Utils::_my_get_edge_num (in 'src/ch_utils.cpp' line 50)

```
uint32_t _my_get_edge_num(node_id_t node_id,
    const CHGraph& graph,
        const std::vector<bool>& node_deleted_flag_vec)
{
    node_descriptor_t node = Utils::node_id2node_descriptor(node_id);

    uint32_t edge_num(0);

    out_edge_iterator_t out_edge_it, out_edge_it_end;
    boost::tie(out_edge_it, out_edge_it_end) = boost::out_edges(node, graph);
    for (; out_edge_it != out_edge_it_end; ++ out_edge_it) {
        node_descriptor_t target = boost::target(*out_edge_it, graph);
        node_id_t target_id = Utils::node_descriptor2node_id(target);

        if (boost::get(boost::edge_weight, graph, *out_edge_it) >= MAX_EDGE_WEIGHT) {
            continue;
        }

        if (node_deleted_flag_vec[target_id]) {
            continue;
        }

        edge_num ++;
    }

    in_edge_iterator_t in_edge_it, in_edge_it_end;
    boost::tie(in_edge_it, in_edge_it_end) = boost::in_edges(node, graph);

    for (; in_edge_it != in_edge_it_end; ++ in_edge_it) {
        node_descriptor_t source = boost::source(*in_edge_it, graph);
        node_id_t source_id = Utils::node_descriptor2node_id(source);

        if (boost::get(boost::edge_weight, graph, *in_edge_it) >= MAX_EDGE_WEIGHT) {
            continue;
        }

        if (node_deleted_flag_vec[source_id]) {
            continue;
        }

        edge_num ++;
    }

    return edge_num;
}
```

输入：某点u，图，已经被删除的点
输出：点u的度数
作用：得到点的度数

node_id和node_descriptor是图的两种格式下点的不同标记方法（不过实际上数值是一模一样的，node_id2node_descriptor实际上什么也没做）

统计出度

有效边未被删

统计入度

有效边未被删

CH 层次图构建

- Utils::calc_edge_diff (in 'src/ch_utils.cpp' line 96)

```
96  int Utils::calc_edge_diff(std::vector<shortcut_info_t> & shortcut_vecs,
97      node_descriptor_t node,
98          const CHGraph& graph,
99          const std::vector<bool>& node_deleted_flag_vec,
100         uint32_t max_hops,
101         int pthread_id)
102 {
103     Utils::find_shortcuts(node,
104             //NOTE: we do no modification in the find_shortcuts function, const_cast
105             //is just for compile purpose
106             const_cast<CHGraph&>(graph),
107             node_deleted_flag_vec,
108             max_hops,
109             shortcut_vecs,
110             pthread_id);
111     // statistic_shortcut_num = shortcut_vecs.size();
112     // statistic_node_in_degree = boost::in_degree(node, graph);
113     // statistic_node_out_degree = boost::out_degree(node, graph);
114
115     /*
116     com_writelog(COMLOG_DEBUG, "node_id:%d", node);
117     com_writelog(COMLOG_DEBUG, "shortcut_vec size:%u", shortcut_vecs.size());
118     com_writelog(COMLOG_DEBUG, "node in degree:%d", boost::in_degree(node, graph));
119     com_writelog(COMLOG_DEBUG, "node out degree:%d", boost::out_degree(node, graph));
120     */
121     int edge_diff = shortcut_vecs.size()*10 - _my_get_edge_num(node, graph, node_deleted_flag_vec);
122         //- boost::in_degree(node, graph)
123         //- boost::out_degree(node, graph);
124
125     return edge_diff;
126 }
```

输入：

...，被contract的点，图，已经被删除的点，局部搜索时的hop限制，并行时的pthread id

输出：

需要添加的shortcut

作用：

计算被contract点的edge difference，顺便进行局部搜索把需要添加的shortcut算出来

进行witness path 局部搜索

计算edge difference，此处为shortcut数*10-点度数

CH 层次图构建

- create_hierarchy_process (in 'src/ch_construct_ch.cpp' line 190)

```
189
190 ch_ret_t ConstructCH::_create_hierarchy_process(CHGraph& graph,
191         std::vector<node_id_t>& node_order_vec,
192         std::vector<shortcut_edge_t>& all_shortcut_vec)
193 {
194     int numVertex = boost::num_vertices(graph);
195     int numEdges = boost::num_edges(graph);
196     CINFO_LOG( "ConstructCH::create_hierarchy_process starts");
197
198     weight_map_t weight_map = boost::get(boost::edge_weight, graph);
199     shortcut_flag_map_t shortcut_flag_map = boost::get(boost::edge_shortcut_flag, graph);
200     node_order_map_t node_order_map = boost::get(boost::vertex_order, graph);
201     edge_id_map_t edge_id_map = boost::get(boost::edge_id, graph);
202
203     EdgeIDAllocator edge_id_allocator(numEdges);
204
205     int shortcuts_added = 0;
206     order_t current_node_order = 0; //record current contract node order
207
```

输入：图
输出：从前到后缩的点，添加的shortcut边
作用：构建CH层次图

边权“数组”（boost::property_map）
是不是shortcut“数组”
第几个被contract“数组”
每条边的id（注意shortcut的id需要新分配）

用来分配新的edge的id

```
209     //debug init;
210     // g_calc_edge_diff_time.reserve(boost::num_vertices(graph));
211     // g_find_shortcut_search_space.reserve(boost::num_vertices(graph));
212
213     for (size_t i = 0; i < boost::num_vertices(graph); ++i) {
214         node_id_t contract_node(0);
215         double myPriority = 0;
216         int reinsert_count = 0; /// debug purpose variable
217         std::vector<shortcut_info_t> shortcut_vec;
218         while (this->_node_pqueue_ptr->empty() != true) {
219             //handle to node
220             node_id_t min_node = this->_node_pqueue_ptr->topKey();
221             double minValue = _node_pqueue_ptr->topValue();
222
223             this->_node_pqueue_ptr->pop();
224
225             if (this->_node_pqueue_ptr->empty()) {
226                 contract_node = min_node;
227                 break;
228             }
229
230             int edge_diff = Utils::calc_edge_diff(
231                 shortcut_vec,
232                 (node_descriptor_t)min_node,
233                     graph,
234                     _node_deleted_flag_vec,
235                     _contract_params.max_hops);
236             _node_edge_diff_vec[min_node] = edge_diff;
237
238             pq_weight_t min_value = Utils::calc_node_priority(
239                     min_node,
240                     _node_edge_diff_vec[min_node],
241                     _node_deleted_neighbours_vec[min_node],
242                     _voronoi_number[min_node],
243                     _contract_params.term_coefficients);
244             myPriority = min_value;
245             node_id_t second_min_node = this->_node_pqueue_ptr->topKey();
246             pq_weight_t second_min_value = this->_node_pqueue_ptr->topValue();
247             if (min_value <= second_min_value) {
248                 contract_node = min_node;
249                 CDEBUG_LOG("reinsert_count:%d", reinsert_count);
250                 break;
251             }
252
253             this->_node_pqueue_ptr->push_or_update(min_node, min_value);
254             reinsert_count ++;
255         }
256
```

Contract主过程
缩n次，缩完为止

拿出当前优先度最小的点
（如果只有一个点，结束）

Lazy update
求最小优先度点

求一个点的
edge difference，
（顺便把shortcut算出来）
然后求其优先度

如果还是最小的，停止；
否则塞回去，重复

CH 层次图构建

- Utils::calc_node_priority (in 'src/ch_utils.cpp' line 33)

```
32
33 double Utils::calc_node_priority(
34         node_id_t node_id,
35         int edge_diff,
36         int deleted_neighbours,
37         double voronoi_region,
38         const std::vector<double>& term_coefficients)
39 {
40     //TODO: remove node_id in the function sig since it is a redundant parameter
41     // double res = edge_diff * 19 +
42     //     deleted_neighbours * 1 +
43     //     sqrt(voronoi_region) * 1;
44     double res = edge_diff * term_coefficients[0] +
45         deleted_neighbours * term_coefficients[1] +
46         sqrt(voronoi_region) * term_coefficients[2];
47
48     return res;
49 }
```

输入：被contract的点u，
点u的edge difference，
点u被contract的邻居数，
点u的voronoi区域中被contract的点数，
各个term的权重

输出：点u的优先度

CH 层次图构建

- Node order（缩点顺序）
  - **Edge difference（边数差）**
  - Cost of contraction（缩点代价）
  - **Uniformity（均匀性）**
  - Cost of queries（查询代价）
  - Global measures（全局特征）

```
209     //debug init;
210     // g_calc_edge_diff_time.reserve(boost::num_vertices(graph));
211     // g_find_shortcut_search_space.reserve(boost::num_vertices(graph));
212
213     for (size_t i = 0; i < boost::num_vertices(graph); ++i) {
214         node_id_t contract_node(0);
215         double myPriority = 0;
216         int reinsert_count = 0; /// debug purpose variable
217         std::vector<shortcut_info_t> shortcut_vec;
218         while (this->_node_pqueue_ptr->empty() != true) {
219             //handle to node
220             node_id_t min_node = this->_node_pqueue_ptr->topKey();
221             double minValue = _node_pqueue_ptr->topValue();
222
223             this->_node_pqueue_ptr->pop();
224
225             if (this->_node_pqueue_ptr->empty()) {
226                 contract_node = min_node;
227                 break;
228             }
229
230             int edge_diff = Utils::calc_edge_diff(
231                 shortcut_vec,
232                 (node_descriptor_t)min_node,
233                     graph,
234                     _node_deleted_flag_vec,
235                     _contract_params.max_hops);
236             _node_edge_diff_vec[min_node] = edge_diff;
237
238             pq_weight_t min_value = Utils::calc_node_priority(
239                     min_node,
240                     _node_edge_diff_vec[min_node],
241                     _node_deleted_neighbours_vec[min_node],
242                     _voronoi_number[min_node],
243                     _contract_params.term_coefficients);
244             myPriority = min_value;
245             node_id_t second_min_node = this->_node_pqueue_ptr->topKey();
246             pq_weight_t second_min_value = this->_node_pqueue_ptr->topValue();
247             if (min_value <= second_min_value) {
248                 contract_node = min_node;
249                 CDEBUG_LOG("reinsert_count:%d", reinsert_count);
250                 break;
251             }
252
253             this->_node_pqueue_ptr->push_or_update(min_node, min_value);
254             reinsert_count ++;
255         }
256
```

Contract主过程
缩n次，缩完为止

拿出当前优先度最小的点
（如果只有一个点，结束）

Lazy update
求最小优先度点

求一个点的
edge difference，
（顺便把shortcut算出来）
然后求其优先度

如果还是最小的，停止；
否则塞回去，重复

Contract主过程

```
257     if (i % _contract_params.debug_trace_cycle == 0) {
258         CINFO_LOG( "%u nodes are contracted", i);
259         CINFO_LOG( "%u shortcut are added", _shortcut_count);
260 
261         uint32_t vertex_num = numVertex;//boost::num_vertices(graph) - i;
262         uint32_t edge_num = numEdges;//boost::num_edges(graph) - _deleted_edge_count;
263         double graph_degree = edge_num * 1.0 / vertex_num;
264 
265         CINFO_LOG( "[nodes: %u] [edges: %u] [graph degree: %f]",
266                 vertex_num,
267                 edge_num,
268                 graph_degree);
269     }
270 
271     node_order_vec.push_back(contract_node);
272     node_order_map[contract_node] = current_node_order;
273     current_node_order ++;
274 
275     //std::vector<shortcut_edge_t> shortcut_vec;
276     //Utils::find_shortcuts(contract_node, graph, _node_deleted_flag_vec, 5, shortcut_vec);
277     shortcuts_added += shortcut_vec.size();
278     //add shortcut edges to graph edge
279     for (size_t i = 0; i < shortcut_vec.size(); ++i) {
280 
281         edge_descriptor_t first_edge = shortcut_vec[i].first_edge;
282         node_descriptor_t start_node = boost::source(first_edge, graph);
283 
284         edge_descriptor_t second_edge = shortcut_vec[i].second_edge;
285         node_descriptor_t end_node = boost::target(second_edge, graph);
286 
287         // set edge ids
288         shortcut_edge_t shortcut;
289         shortcut.first_edge_id = edge_id_map[first_edge];
290         shortcut.second_edge_id = edge_id_map[second_edge];
291         shortcut.id = edge_id_allocator.get_next_id();
292         all_shortcut_vec.push_back(shortcut);
293 
294         edge_descriptor_t shortcut_edge;
295         bool flag;
296         boost::tie(shortcut_edge, flag) = boost::add_edge(start_node, end_node, graph);
297         weight_map[shortcut_edge] = shortcut_vec[i].weight;
298         shortcut_flag_map[shortcut_edge] = true;
299         edge_id_map[shortcut_edge] = shortcut.id;
300 
301         CDEBUG_LOG("add shortcut [start_node:%u] [end_node:%u]", start_node, end_node);
302     }
303 
304     // record the shortcus
305 
306     // all_shortcut_vec.insert(all_shortcut_vec.end(),
307     //     shortcut_vec.begin(), shortcut_vec.end());
308 
309     numEdges += shortcut_vec.size();
310 
311     //remove in-edges and out-edges of node v from graph
312     // mark node as deleted
313     _node_deleted_flag_vec[contract_node] = true;
314 
315     _deleted_edge_count += _get_edge_num(contract_node, graph);
316     _shortcut_count += shortcut_vec.size();
317 
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320 
```

输出debug信息

更新contract node和shortcut相关的信息

比如shortcut是v->u->w

点v

点w

真正地把shortcut都加到图里面
(刚算出来的时候只是存在数组里)

设置一堆id属性：
v->u的id是啥
u->w的id是啥
v->w的id是啥

设权重/shortcut标记/新添shortcut v->w的id

总边数更新

contract掉的点对整个图的影响

这个点被contract掉了

删掉了这么多边
加上了这么多边

理论上是更新voronoi信息，实际上是什么都没做.....

CH 层次图构建

- _distribute_voronoi_region (in 'src/ch_construct_ch.cpp' line 591)

```
591 ch_ret_t ConstructCH::_distribute_voronoi_region(node_id_t node_id, CHGraph& graph)
592 {
593     return CH_RET_OK;
594     CDEBUG_LOG("ConstructCH::distribute_voronoi_region starts");
595     // The nodes of the Voronoi region are stored in a single linked list that
596     // is terminated by SPECIAL_NODEID. The DijkstraCH class provides
597     // only low level methods to update/insert nodes and extract The
598     // node with the lowest distance. The main logic is in this subroutine.
599     _voronoi_pqueue.clear();
600     node_id_t current = node_id;
601
602     while ( current != SPECIAL_NODEID )
603     {
604         // Under all nodes that are incident to an incoming node of the currently
605         // regarded node in R(node), and that are not in R(node), take the node
```

输入：被contract的点u，图
作用：理论上是分配点u的voronoi区域，实际上是什么都不做

Contract主过程

```
257     if (i % _contract_params.debug_trace_cycle == 0) {
258         CINFO_LOG( "%u nodes are contracted", i);
259         CINFO_LOG( "%u shortcut are added", _shortcut_count);
260 
261         uint32_t vertex_num = numVertex;//boost::num_vertices(graph) - i;
262         uint32_t edge_num = numEdges;//boost::num_edges(graph) - _deleted_edge_count;
263         double graph_degree = edge_num * 1.0 / vertex_num;
264 
265         CINFO_LOG( "[nodes: %u] [edges: %u] [graph degree: %f]",
266                 vertex_num,
267                 edge_num,
268                 graph_degree);
269     }
270 
271     node_order_vec.push_back(contract_node);
272     node_order_map[contract_node] = current_node_order;
273     current_node_order ++;
274 
275     //std::vector<shortcut_edge_t> shortcut_vec;
276     //Utils::find_shortcuts(contract_node, graph, _node_deleted_flag_vec, 5, shortcut_vec);
277     shortcuts_added += shortcut_vec.size();
278     //add shortcut edges to graph edge
279     for (size_t i = 0; i < shortcut_vec.size(); ++i) {
280 
281         edge_descriptor_t first_edge = shortcut_vec[i].first_edge;
282         node_descriptor_t start_node = boost::source(first_edge, graph);
283 
284         edge_descriptor_t second_edge = shortcut_vec[i].second_edge;
285         node_descriptor_t end_node = boost::target(second_edge, graph);
286 
287         // set edge ids
288         shortcut_edge_t shortcut;
289         shortcut.first_edge_id = edge_id_map[first_edge];
290         shortcut.second_edge_id = edge_id_map[second_edge];
291         shortcut.id = edge_id_allocator.get_next_id();
292         all_shortcut_vec.push_back(shortcut);
293 
294         edge_descriptor_t shortcut_edge;
295         bool flag;
296         boost::tie(shortcut_edge, flag) = boost::add_edge(start_node, end_node, graph);
297         weight_map[shortcut_edge] = shortcut_vec[i].weight;
298         shortcut_flag_map[shortcut_edge] = true;
299         edge_id_map[shortcut_edge] = shortcut.id;
300 
301         CDEBUG_LOG("add shortcut [start_node:%u] [end_node:%u]", start_node, end_node);
302     }
303 
304     // record the shortcus
305 
306     // all_shortcut_vec.insert(all_shortcut_vec.end(),
307     //     shortcut_vec.begin(), shortcut_vec.end());
308 
309     numEdges += shortcut_vec.size();
310 
311     //remove in-edges and out-edges of node v from graph
312     // mark node as deleted
313     _node_deleted_flag_vec[contract_node] = true;
314 
315     _deleted_edge_count += _get_edge_num(contract_node, graph);
316     _shortcut_count += shortcut_vec.size();
317 
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320 
```

输出debug信息

更新contract node和shortcut相关的信息

比如shortcut是v->u->w

点v

点w

真正地把shortcut都加到图里面
(刚算出来的时候只是存在数组里)

设置一堆id属性：
v->u的id是啥
u->w的id是啥
v->w的id是啥

设权重/shortcut标记/新添shortcut v->w的id

总边数更新

contract掉的点对整个图的影响

这个点被contract掉了

删掉了这么多边
加上了这么多边

理论上是更新voronoi信息，实际上是什么都没做.....

```
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320
321     int goodEdge = 0;
322     //update neighbours priority
323     //adjacent vertices update
324     //com_writelog(COMLOG_DEBUG, "update neighbours priority starts");
325     std::pair<adjacency_iterator, adjacency_iterator> adj_vertices =
326         boost::adjacent_vertices(contract_node, graph);
327
328     for ( ;adj_vertices.first != adj_vertices.second; ++adj_vertices.first) {
329         node_id_t node_id = *(adj_vertices.first);
330
331         if (_node_deleted_flag_vec[node_id] == true) {
332             continue;
333         }
334         goodEdge ++;
335         this->_update_node_priority(node_id, graph);
336     }
337
338     //inv adjacent vertices update
339     std::pair<inv_adjacency_iterator, inv_adjacency_iterator> inv_adj_vertices =
340         boost::inv_adjacent_vertices(contract_node, graph);
341
342     for ( ;inv_adj_vertices.first != inv_adj_vertices.second; ++inv_adj_vertices.first) {
343         node_id_t node_id = *(inv_adj_vertices.first);
344
345         if (_node_deleted_flag_vec[node_id] == true) {
346             continue;
347         }
348         goodEdge ++;
349         this->_update_node_priority(node_id, graph);
350     }
351     //com_writelog(COMLOG_DEBUG, "update neighbours priority ends");
352
353     numEdges -= goodEdge;
354     numVertex --;
355     ///dump debug info
377     if (this->_global_update_required(numVertex, numEdges)) {
378         CINFO_LOG( "start global update");
379
380         // CINFO_LOG("before parallel\ndel_flag:%llx\ndel_nei:%llx\ne_diff:%llx,cp:%llx"
381         //注释&_node_deleted_flag_vec,
382         //     &_node_deleted_neighbours_vec,
383         //     &_node_edge_diff_vec,
384         //     &_contract_params);
385
386         Utils::global_update(graph,
387             _node_deleted_flag_vec,
388             _node_deleted_neighbours_vec,
389             _voronoi_number,
390             *_node_pqueue_ptr,
391             _node_edge_diff_vec,
392             _contract_params);
467         //注释
468         CINFO_LOG( "end global update");
469     }
470 }
471
472 /*if (com_log_enabled(COMLOG_DEBUG)) {
473     _debug_stream.close();
474 }*/
475 std::cout << "shortcuts_added=" << shortcuts_added << std::endl;
476 CINFO_LOG( "ConstructCH::create_hierarchy_process ends");
477 return CH_RET_OK;
478 }
```

Contract
主过程

Update
邻居节点

删边删点

全局Update

CH 层次图构建

- _update_node_priority (in 'src/ch_construct_ch.cpp' line 120)

```
120 ch_ret_t  ConstructCH::_update_node_priority(node_id_t node_id,
121     CHGraph& graph)
122 {
123     /*
124     //update edge diff
125     int edge_diff = Utils::calc_edge_diff(node_id,
126             graph,
127             _node_deleted_flag_vec,
128             5);
129     _node_edge_diff_vec[node_id] = edge_diff;
130     */
131     //update DeletedNeighbours
132     ++ _node_deleted_neighbours_vec[node_id];
133
134     //recalc priority and update position
135     double priority = Utils::calc_node_priority(
136         node_id,
137         _node_edge_diff_vec[node_id],
138         _node_deleted_neighbours_vec[node_id],
139         _voronoi_number[node_id],
140         _contract_params.term_coefficients);
141
142     this->_node_pqueue_ptr->push_or_update(node_id, priority);
143     return CH_RET_OK;
144 }
```

输入：某个点u，图
作用：在点u的邻居被contract后更新点u的优先度

更新点u的term：被contract的邻居数

重新计算点u的优先度

更新优先队列中点u的优先度

```
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320
321     int goodEdge = 0;
322     //update neighbours priority
323     //adjacent vertices update
324     //com_writelog(COMLOG_DEBUG, "update neighbours priority starts");
325     std::pair<adjacency_iterator, adjacency_iterator> adj_vertices =
326         boost::adjacent_vertices(contract_node, graph);
327
328     for ( ;adj_vertices.first != adj_vertices.second; ++adj_vertices.first) {
329         node_id_t node_id = *(adj_vertices.first);
330
331         if (_node_deleted_flag_vec[node_id] == true) {
332             continue;
333         }
334         goodEdge ++;
335         this->_update_node_priority(node_id, graph);
336     }
337
338     //inv adjacent vertices update
339     std::pair<inv_adjacency_iterator, inv_adjacency_iterator> inv_adj_vertices =
340         boost::inv_adjacent_vertices(contract_node, graph);
341
342     for ( ;inv_adj_vertices.first != inv_adj_vertices.second; ++inv_adj_vertices.first) {
343         node_id_t node_id = *(inv_adj_vertices.first);
344
345         if (_node_deleted_flag_vec[node_id] == true) {
346             continue;
347         }
348         goodEdge ++;
349         this->_update_node_priority(node_id, graph);
350     }
351     //com_writelog(COMLOG_DEBUG, "update neighbours priority ends");
352
353     numEdges -= goodEdge;
354     numVertex --;
355     ///dump debug info
377     if (this->_global_update_required(numVertex, numEdges)) {
378         CINFO_LOG( "start global update");
379
380         // CINFO_LOG("before parallel\ndel_flag:%llx\ndel_nei:%llx\ne_diff:%llx,cp:%llx"
381         //注释&_node_deleted_flag_vec,
382         //     &_node_deleted_neighbours_vec,
383         //     &_node_edge_diff_vec,
384         //     &_contract_params);
385
386         Utils::global_update(graph,
387             _node_deleted_flag_vec,
388             _node_deleted_neighbours_vec,
389             _voronoi_number,
390             *_node_pqueue_ptr,
391             _node_edge_diff_vec,
392             _contract_params);
467         注释
468         CINFO_LOG( "end global update");
469     }
470 }
471
472 /*if (com_log_enabled(COMLOG_DEBUG)) {
473     _debug_stream.close();
474 }*/
475 std::cout << "shortcuts_added=" << shortcuts_added << std::endl;
476 CINFO_LOG( "ConstructCH::create_hierarchy_process ends");
477 return CH_RET_OK;
478 }
```

Contract
主过程

Update
邻居节点

删边删点

全局Update

CH 层次图构建

- _global_update_required (in 'src/ch_construct_ch.cpp' line 33)

```
32
33 bool ConstructCH::_global_update_required(int numVertex, int numEdges)
34 {
35     if (_current_level >= this->_contract_params.update_degrees.size()) {
36         return false;
37     }
38
39     double avg_degree = numEdges * 1.0 / numVertex;
40     if (avg_degree > this->_contract_params.update_degrees[_current_level]) {
41         _current_level ++;
42         CINFO_LOG( "gloabl update, [nodes:%u] [edges:%u] [avg degree:%f]",
43                 numVertex, numEdges, avg_degree);
44         return true;
45     }
46
47     return false;
48 }
```

输入：图的点数，图的边数
输出：是否需要进行全局优先度更新

使用平均度数确定是否需要进行全局优先度更新

首先确定一个单调上升数组，比如{3，5，8，10，...}。之后当平均度数超过当前数值后进行全局优先度更新，并将当前数值设为数组中的下一个数值

```
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320
321     int goodEdge = 0;
322     //update neighbours priority
323     //adjacent vertices update
324     //com_writelog(COMLOG_DEBUG, "update neighbours priority starts");
325     std::pair<adjacency_iterator, adjacency_iterator> adj_vertices =
326         boost::adjacent_vertices(contract_node, graph);
327
328     for ( ;adj_vertices.first != adj_vertices.second; ++adj_vertices.first) {
329         node_id_t node_id = *(adj_vertices.first);
330
331         if (_node_deleted_flag_vec[node_id] == true) {
332             continue;
333         }
334         goodEdge ++;
335         this->_update_node_priority(node_id, graph);
336     }
337
338     //inv adjacent vertices update
339     std::pair<inv_adjacency_iterator, inv_adjacency_iterator> inv_adj_vertices =
340         boost::inv_adjacent_vertices(contract_node, graph);
341
342     for ( ;inv_adj_vertices.first != inv_adj_vertices.second; ++inv_adj_vertices.first) {
343         node_id_t node_id = *(inv_adj_vertices.first);
344
345         if (_node_deleted_flag_vec[node_id] == true) {
346             continue;
347         }
348         goodEdge ++;
349         this->_update_node_priority(node_id, graph);
350     }
351     //com_writelog(COMLOG_DEBUG, "update neighbours priority ends");
352
353     numEdges -= goodEdge;
354     numVertex --;
355     ///dump debug info
377     if (this->_global_update_required(numVertex, numEdges)) {
378         CINFO_LOG( "start global update");
379
380         // CINFO_LOG("before parallel\ndel_flag:%llx\ndel_nei:%llx\ne_diff:%llx,cp:%llx"
381         //注释&_node_deleted_flag_vec,
382         //     &_node_deleted_neighbours_vec,
383         //     &_node_edge_diff_vec,
384         //     &_contract_params);
385
386         Utils::global_update(graph,
387             _node_deleted_flag_vec,
388             _node_deleted_neighbours_vec,
389             _voronoi_number,
390             *_node_pqueue_ptr,
391             _node_edge_diff_vec,
392             _contract_params);
467         注释
468         CINFO_LOG( "end global update");
469     }
470 }
471
472 /*if (com_log_enabled(COMLOG_DEBUG)) {
473     _debug_stream.close();
474 }*/
475 std::cout << "shortcuts_added=" << shortcuts_added << std::endl;
476 CINFO_LOG( "ConstructCH::create_hierarchy_process ends");
477 return CH_RET_OK;
478 }
```

Contract
主过程

Update
邻居节点

删边删点

全局Update

CH 层次图构建

- Utils::global_update (in 'src/ch_utils.cpp' line 1025)

```
1024
1025 void Utils::global_update(const CHGraph& graph,
1026     const std::vector<bool> &_node_deleted_flag_vec,
1027     const std::vector<uint32_t>& _node_deleted_neighbours_vec,
1028     const std::vector<uint32_t> &_voronoi_number,
1029     NodePriorityQueue& _node_pqueue,
1030     std::vector<int32_t>& _node_edge_diff_vec,
1031     const ContractParameters &contract_paramter)
1032 {
1033     // printf("start global update\n");
1034     // Utils::stat_max_open_table_size=0;
1035     // Utils::stat_max_close_table_size=0;
1036     // Utils::stat_max_sources_size=0;
1037     // Utils::stat_max_targets_size=0;
1038
1039     Timer timer;
1040     Utils::log_mem_usage("global update");
1041     Utils::pthread_master(graph,
1042         _node_deleted_flag_vec, _node_deleted_neighbours_vec, _voronoi_number,
1043         _node_pqueue, _node_edge_diff_vec, contract_paramter);
1044     uint32_t t = timer.get_time_ms();
1045     CINFO_LOG("global update time:%u",t);
1046     Utils::statistic_global_update_time += t;
1047
1048     // CINFO_LOG("local search size stat: [open:%d] , [close:%d] , [source:%d], [targets:%d]",
1049     //     Utils::stat_max_open_table_size,
1050     //     Utils::stat_max_close_table_size,
1051     //     Utils::stat_max_sources_size,
1052     //     Utils::stat_max_targets_size);
1053 }
```

输入：图，
计算优先度需要的terms，
计算优先度最小点时需要用到的优先队列，
各个点初始的edge difference
contract的参数（比如各个term的权重什么的）
输出：优先队列需要初始化（现在是空的），
edge difference需要初始化，
作用：更新所有点的contract优先度

多线程并行计算点的contract优先度

计时看看要多久

CH 层次图构建

- Utils::pthread_master (in 'src/ch_utils.cpp' line 972)

```
968     pthread_barrier_wait(&barrier_master);
969     return (void*)0;
970 }
971
972 void Utils::pthread_master(const CHGraph& graph,
973     const std::vector<bool>& _node_deleted_flag_vec,
974     const std::vector<uint32_t>& _node_deleted_neighbours_vec,
975     const std::vector<uint32_t> &_voronoi_number,
976     NodePriorityQueue& _node_pqueue,
977     std::vector<int32_t> &_node_edge_diff_vec,
978     const ContractParameters &contract_paramter)
979 {
980     if (contract_paramter.thread_num > 1 && contract_paramter.max_hops < 3) {
981         // use static data in _fast_local_search.
982         CWARNING_LOG("parallelize global update while max_hops < 3 is not supported!! Do some update in '_fast_local_search'!");
983         CWARNING_LOG("set thread_num to 1!");
984         Utils::thread_num = 1;
985     } else {
986         Utils::thread_num = contract_paramter.thread_num;
987     }
988     if(Utils::thread_num > MAX_THREAD_NUM) {
989         Utils::thread_num = MAX_THREAD_NUM;
990     }
991
992     pthread_barrier_init(&Utils::barrier_master, NULL, Utils::thread_num+1);
993     pthread_mutex_init(&Utils::worker_mtx, NULL);
994     int node_num = boost::num_vertices(graph);
995
996     Utils::thread_task_size = (node_num  + contract_paramter.task_unit_num - 1) / contract_paramter.task_unit_num ;
997     PthreadArg arg[MAX_THREAD_NUM];
998     Utils::working_process = 0;
999     for(int i = 0; i < thread_num; i++) {
1000         arg[i].arg_graph = &graph;
1001         arg[i].arg_node_edge_diff_vec = &_node_edge_diff_vec;
1002         arg[i].arg_node_deleted_flag_vec = &_node_deleted_flag_vec;
1003         arg[i].arg_node_deleted_neighbours_vec = &_node_deleted_neighbours_vec;
1004         arg[i].arg_contract_paramter = &contract_paramter;
1005         arg[i].arg_voronoi_number = &_voronoi_number;
1006         arg[i].arg_pthread_id = i;
1007         pthread_create(&(threads[i]), NULL, Utils::pthread_worker, &arg[i]);
1008
1009     pthread_barrier_wait(&Utils::barrier_master);
1010
1011     CINFO_LOG("all thread worker done");
1012     std::pair<node_iterator_t, node_iterator_t> node_range = boost::vertices(graph);
1013     for (node_iterator_t cur_node = node_range.first; cur_node != node_range.second; ++ cur_node) {
1014         node_id_t cur_node_id = (node_id_t)(*cur_node);
1015         if (!_node_deleted_flag_vec[cur_node_id]) {
1016             //printf("%u %u\n",cur_node_id, Utils::node_pvector[cur_node_id]);
1017             _node_pqueue.push_or_update(cur_node_id, Utils::node_pvector[cur_node_id]);
1018         }
1019     }
1020
1021     pthread_barrier_destroy(&Utils::barrier_master);
1022     pthread_mutex_destroy(&Utils::worker_mtx);
1023 }
```

输入：图，
计算优先度需要的3个terms，
计算优先度最小点时需要用到的优先队列
各个点初始的edge difference
contract的参数（比如各个term的权重什么的）
输出：优先队列需要重新得到，
edge difference需要重新得到
作用：更新所有点的contract优先度

当前_fast_local_search里面
没有将数据结构按thread分开，
所以工程上还没有支持，
理论上是可以支持的

barrier和互斥锁初始化

将任务分成
固定份数，
然后产生几个
thread完成
任务

并行的方式是
哪个thread
做完了就去领
下一份任务

初始化几个thread，
给定必要的参数

更新计算最小优先度点
的优先队列

CH 层次图构建

- Utils::pthread_worker (in 'src/ch_utils.cpp' line 907)

输入：更新优先度需要的所有参数
作用：更新一部分点（分配得到的任务范围内）的优先度

```
905
906
907 void * Utils::pthread_worker(void *arg) {
908     PthreadArg parg = *((PthreadArg*)arg);
909     int pthread_id = parg.arg_pthread_id;
910     const CHGraph& graph = *(parg.arg_graph);
911     const std::vector<bool>& _node_deleted_flag_vec = *(parg.arg_node_deleted_flag_vec);
912     const std::vector<uint32_t>& _node_deleted_neighbours_vec = *(parg.arg_node_deleted_neighbours_vec);
913     std::vector<int32_t>& _node_edge_diff_vec = *(parg.arg_node_edge_diff_vec);
914     const ContractParameters &contract_paramter = *(parg.arg_contract_paramter);
915     const std::vector<uint32_t> &_voronoi_number = *(parg.arg_voronoi_number);
916
917     //□ □ CPU□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
918     cpu_set_t mask;
919     CPU_ZERO(&mask);
920     CPU_SET(pthread_id, &mask);
921     if(sched_setaffinity(0, sizeof(mask), &mask) == -1) {//0 □ □ □ □ □ □ □ /□ □ □ □ □ □ □ .
922         CINFO_LOG("set affinity failed..");
923     }
924
925
926     int start ;//= pthread_id * Utils::thread_task_size;
927     CINFO_LOG("thread worker id=%u, starts",pthread_id);
928     uint32_t node_num = boost::num_vertices(graph);
929     uint32_t cnt = 0;
930     while (true) {
931         pthread_mutex_lock(&Utils::worker_mtx);
932         if (Utils::working_process >= node_num) {
933             pthread_mutex_unlock(&Utils::worker_mtx);
934             break;
935         } else {
936             start = Utils::working_process;
937             Utils::working process += Utils::thread task size;
938         }
939         pthread_mutex_unlock(&Utils::worker_mtx);
940
941         for (int i = 0 ; i < Utils::thread_task_size; i++){
942             // node_id_t cur_node_id = (node_id_t)(*cur_node);
943             node_id_t cur_node_id = start+i;
944             if (cur_node_id >= node_num) break;
945             if(_node_deleted_flag_vec[cur_node_id]) {
946                 continue;
947             }
948             cnt ++;
949             std::vector<shortcut_info_t> shortcut_vec;
950             int edge_diff = Utils::calc_edge_diff(
951                 shortcut_vec,
952                 cur_node_id,
953                 graph,
954                 _node_deleted_flag_vec,
955                 contract_paramter.max_hops,
956                 pthread_id);
957             _node_edge_diff_vec[cur_node_id] = edge_diff;
958             double priority = Utils::calc_node_priority(cur_node_id,
959                         _node_edge_diff_vec[cur_node_id],
960                         _node_deleted_neighbours_vec[cur_node_id],
961                         _voronoi_number[cur_node_id],
962                         contract_paramter.term_coefficients);
963             Utils::node_pvector[cur_node_id] = priority;
964
965         }
966     }
967     CINFO_LOG("thread worker done! id=%u, cnt=%u",pthread_id,cnt);
968     pthread_barrier_wait(&barrier_master);
969     return (void*)0;
970 }
```

解析参数：
图
已经删除的点
删除的邻居数
edge difference
contract的参数
（比如term的权重）
voronoi区域中的点数

设置CPU亲和性，
提高缓存命中率

利用互斥锁
领取配额，
改变任务进度

领取任务配额
并工作

完成指定的任务配额，
即更新给定的一部分点
的优先度

更新edge difference

更新优先度

CH 层次图构建

- create_hierarchy_process (in 'src/ch_construct_ch.cpp' line 190)

```
189
190 ch_ret_t ConstructCH::_create_hierarchy_process(CHGraph& graph,
191         std::vector<node_id_t>& node_order_vec,
192         std::vector<shortcut_edge_t>& all_shortcut_vec)
193 {
194     int numVertex = boost::num_vertices(graph);
195     int numEdges = boost::num_edges(graph);
196     CINFO_LOG( "ConstructCH::create_hierarchy_process starts");
197
198     weight_map_t weight_map = boost::get(boost::edge_weight, graph);
199     shortcut_flag_map_t shortcut_flag_map = boost::get(boost::edge_shortcut_flag, graph);
200     node_order_map_t node_order_map = boost::get(boost::vertex_order, graph);
201     edge_id_map_t edge_id_map = boost::get(boost::edge_id, graph);
202
203     EdgeIDAllocator edge_id_allocator(numEdges);
204
205     int shortcuts_added = 0;
206     order_t current_node_order = 0; //record current contract node order
207
```

输入：图
输出：从前到后缩的点，添加的shortcut边
作用：构建CH层次图

边权“数组”（boost::property_map）
是不是shortcut“数组”
第几个被contract“数组”
每条边的id（注意shortcut的id需要新分配）

用来分配新的edge的id

```
209     //debug init;
210     // g_calc_edge_diff_time.reserve(boost::num_vertices(graph));
211     // g_find_shortcut_search_space.reserve(boost::num_vertices(graph));
212
213     for (size_t i = 0; i < boost::num_vertices(graph); ++i) {
214         node_id_t contract_node(0);
215         double myPriority = 0;
216         int reinsert_count = 0; /// debug purpose variable
217         std::vector<shortcut_info_t> shortcut_vec;
218         while (this->_node_pqueue_ptr->empty() != true) {
219             //handle to node
220             node_id_t min_node = this->_node_pqueue_ptr->topKey();
221             double minValue = _node_pqueue_ptr->topValue();
222
223             this->_node_pqueue_ptr->pop();
224
225             if (this->_node_pqueue_ptr->empty()) {
226                 contract_node = min_node;
227                 break;
228             }
229
230             int edge_diff = Utils::calc_edge_diff(
231                 shortcut_vec,
232                 (node_descriptor_t)min_node,
233                     graph,
234                     _node_deleted_flag_vec,
235                     _contract_params.max_hops);
236             _node_edge_diff_vec[min_node] = edge_diff;
237
238             pq_weight_t min_value = Utils::calc_node_priority(
239                     min_node,
240                     _node_edge_diff_vec[min_node],
241                     _node_deleted_neighbours_vec[min_node],
242                     _voronoi_number[min_node],
243                     _contract_params.term_coefficients);
244             myPriority = min_value;
245             node_id_t second_min_node = this->_node_pqueue_ptr->topKey();
246             pq_weight_t second_min_value = this->_node_pqueue_ptr->topValue();
247             if (min_value <= second_min_value) {
248                 contract_node = min_node;
249                 CDEBUG_LOG("reinsert_count:%d", reinsert_count);
250                 break;
251             }
252
253             this->_node_pqueue_ptr->push_or_update(min_node, min_value);
254             reinsert_count ++;
255         }
256
```

Contract主过程
缩n次，缩完为止

拿出当前优先度最小的点
（如果只有一个点，结束）

Lazy update
求最小优先度点

求一个点的
edge difference，
（顺便把shortcut算出来）
然后求其优先度

如果还是最小的，停止；
否则塞回去，重复

Contract主过程

```
257     if (i % _contract_params.debug_trace_cycle == 0) {
258         CINFO_LOG( "%u nodes are contracted", i);
259         CINFO_LOG( "%u shortcut are added", _shortcut_count);
260 
261         uint32_t vertex_num = numVertex;//boost::num_vertices(graph) - i;
262         uint32_t edge_num = numEdges;//boost::num_edges(graph) - _deleted_edge_count;
263         double graph_degree = edge_num * 1.0 / vertex_num;
264 
265         CINFO_LOG( "[nodes: %u] [edges: %u] [graph degree: %f]",
266                 vertex_num,
267                 edge_num,
268                 graph_degree);
269     }
270 
271     node_order_vec.push_back(contract_node);
272     node_order_map[contract_node] = current_node_order;
273     current_node_order ++;
274 
275     //std::vector<shortcut_edge_t> shortcut_vec;
276     //Utils::find_shortcuts(contract_node, graph, _node_deleted_flag_vec, 5, shortcut_vec);
277     shortcuts_added += shortcut_vec.size();
278     //add shortcut edges to graph edge
279     for (size_t i = 0; i < shortcut_vec.size(); ++i) {
280 
281         edge_descriptor_t first_edge = shortcut_vec[i].first_edge;
282         node_descriptor_t start_node = boost::source(first_edge, graph);
283 
284         edge_descriptor_t second_edge = shortcut_vec[i].second_edge;
285         node_descriptor_t end_node = boost::target(second_edge, graph);
286 
287         // set edge ids
288         shortcut_edge_t shortcut;
289         shortcut.first_edge_id = edge_id_map[first_edge];
290         shortcut.second_edge_id = edge_id_map[second_edge];
291         shortcut.id = edge_id_allocator.get_next_id();
292         all_shortcut_vec.push_back(shortcut);
293 
294         edge_descriptor_t shortcut_edge;
295         bool flag;
296         boost::tie(shortcut_edge, flag) = boost::add_edge(start_node, end_node, graph);
297         weight_map[shortcut_edge] = shortcut_vec[i].weight;
298         shortcut_flag_map[shortcut_edge] = true;
299         edge_id_map[shortcut_edge] = shortcut.id;
300 
301         CDEBUG_LOG("add shortcut [start_node:%u] [end_node:%u]", start_node, end_node);
302     }
303 
304     // record the shortcus
305 
306     // all_shortcut_vec.insert(all_shortcut_vec.end(),
307     //     shortcut_vec.begin(), shortcut_vec.end());
308 
309     numEdges += shortcut_vec.size();
310 
311     //remove in-edges and out-edges of node v from graph
312     // mark node as deleted
313     _node_deleted_flag_vec[contract_node] = true;
314 
315     _deleted_edge_count += _get_edge_num(contract_node, graph);
316     _shortcut_count += shortcut_vec.size();
317 
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320 
```

输出debug信息

更新contract node和shortcut相关的信息

比如shortcut是v->u->w

点v

点w

真正地把shortcut都加到图里面
(刚算出来的时候只是存在数组里)

设置一堆id属性：
v->u的id是啥
u->w的id是啥
v->w的id是啥

设权重/shortcut标记/新添shortcut v->w的id

总边数更新

contract掉的点对整个图的影响

这个点被contract掉了

删掉了这么多边
加上了这么多边

理论上是更新voronoi信息，实际上是什么都没做.....

```
318     //distribute voronoi region
319     this->_distribute_voronoi_region(contract_node, graph);
320
321     int goodEdge = 0;
322     //update neighbours priority
323     //adjacent vertices update
324     //com_writelog(COMLOG_DEBUG, "update neighbours priority starts");
325     std::pair<adjacency_iterator, adjacency_iterator> adj_vertices =
326         boost::adjacent_vertices(contract_node, graph);
327
328     for ( ;adj_vertices.first != adj_vertices.second; ++adj_vertices.first) {
329         node_id_t node_id = *(adj_vertices.first);
330
331         if (_node_deleted_flag_vec[node_id] == true) {
332             continue;
333         }
334         goodEdge ++;
335         this->_update_node_priority(node_id, graph);
336     }
337
338     //inv adjacent vertices update
339     std::pair<inv_adjacency_iterator, inv_adjacency_iterator> inv_adj_vertices =
340         boost::inv_adjacent_vertices(contract_node, graph);
341
342     for ( ;inv_adj_vertices.first != inv_adj_vertices.second; ++inv_adj_vertices.first) {
343         node_id_t node_id = *(inv_adj_vertices.first);
344
345         if (_node_deleted_flag_vec[node_id] == true) {
346             continue;
347         }
348         goodEdge ++;
349         this->_update_node_priority(node_id, graph);
350     }
351     //com_writelog(COMLOG_DEBUG, "update neighbours priority ends");
352
353     numEdges -= goodEdge;
354     numVertex --;
355     ///dump debug info
377     if (this->_global_update_required(numVertex, numEdges)) {
378         CINFO_LOG( "start global update");
379
380         // CINFO_LOG("before parallel\ndel_flag:%llx\ndel_nei:%llx\ne_diff:%llx,cp:%llx"
381         //注释&_node_deleted_flag_vec,
382         //     &_node_deleted_neighbours_vec,
383         //     &_node_edge_diff_vec,
384         //     &_contract_params);
385
386         Utils::global_update(graph,
387             _node_deleted_flag_vec,
388             _node_deleted_neighbours_vec,
389             _voronoi_number,
390             *_node_pqueue_ptr,
391             _node_edge_diff_vec,
392             _contract_params);
467         //注释
468         CINFO_LOG( "end global update");
469     }
470 }
471
472 /*if (com_log_enabled(COMLOG_DEBUG)) {
473     _debug_stream.close();
474 }*/
475 std::cout << "shortcuts_added=" << shortcuts_added << std::endl;
476 CINFO_LOG( "ConstructCH::create_hierarchy_process ends");
477 return CH_RET_OK;
478 }
```

Contract
主过程

Update
邻居节点

删边删点

全局Update

CH 层次图构建

- create_hierarchy (in 'src/ch_construct_ch.cpp' line 480)

```
480 ch_ret_t ConstructCH::create_hierarchy(CHGraph& graph,
481         std::vector<node_id_t>& node_order_vec,
482         std::vector<shortcut_edge_t>& shortcut_edge_vec)
483 {
484     CINFO_LOG( "ConstructCH::create_hierarchy starts");
485
486     int node_num = boost::num_vertices(graph);
487     Utils::init_thread_data(node_num,_contract_params.thread_num);
488
489     //split the function as two subroutines just for readablity consideration
490     this->_create_hierarchy_init(graph, shortcut_edge_vec);
491     this->_create_hierarchy_process(graph,
492             node_order_vec,
493             shortcut_edge_vec);
494
495     Utils::destroy_thread_data(_contract_params.thread_num);
496     CINFO_LOG( "ConstructCH::create_hierarchy ends");
497     return CH_RET_OK;
498 }
```

输入：一个图
输出：从前到后缩的点，添加的shortcut边
作用：根据给定图构建CH层次图

层次图构建过程中有并行操作，需要使用多线程，所以这里进行线程数据的初始化

各种数据初始化，主要是初始化了各个点的contract优先度

层次图构建主要过程

销毁线程数据

CH 层次图构建

- create_hierarchy_init (in 'src/ch_construct_ch.cpp' line 50)

```
50 ch_ret_t ConstructCH::_create_hierarchy_init(CHGraph& graph,
51         std::vector<shortcut_edge_t>& all_shortcut_vec)
52 {
53     CINFO_LOG( "ConstructCH::create_hierarchy_init starts");
54     //init
55     int node_num = boost::num_vertices(graph);
56
57     _node_deleted_flag_vec.resize(node_num, false);
58     _node_deleted_neighbours_vec.resize(node_num, 0);
59     _node_edge_diff_vec.resize(node_num, 0);
60     _voronoi_number.resize(node_num, 1);
61
62     // calc initial pqueue
63     CINFO_LOG( "pqueue init starts");
64      node_pqueue_ptr->init(node_num,node_num);
65     CINFO_LOG( "start global update");
66     Utils::global_update(graph,
67                  _node_deleted_flag_vec,
68                  _node_deleted_neighbours_vec,
69                  _voronoi_number,
70                  *_node_pqueue_ptr,
71                  _node_edge_diff_vec,
72                  _contract_params);
73     CINFO_LOG( "end global update");
92       被注释掉的代码......
93     CINFO_LOG( "pqueue init ends");
94
95     _current_level = 0;
96     _deleted_edge_count = 0;
97     _shortcut_count = 0;
98
114      被注释掉的代码......
115     all_shortcut_vec.reserve(MAX_SHORTCUT_NUM);
116     CINFO_LOG( "ConstructCH::create_hierarchy_init ends");
117     return CH_RET_OK;
118 }
```

输入：当前图，需要保留的所有shortcut（空）
输出：没有
作用：初始化构建层次图需要的数据结构，主要需要初始化各个点的contract优先度

初始化：
“点是否被缩掉”数组
“几个邻居被缩掉”数组
“点的edge difference”数组
“点的voronoi区域中有几个点”数组

初始化：
用来确定最小优先度的优先队列

更新所有点的contract优先度

初始化：
当前全局更新几次了=0
当前总共删掉几条边了=0(实际上这个变量没用)
当前加上几条shortcut边了=0

初始化：
所有的要加的shortcut，初始为空，但预留一定空间

CH 层次图构建

- create_hierarchy (in 'src/ch_construct_ch.cpp' line 480)

```
480 ch_ret_t ConstructCH::create_hierarchy(CHGraph& graph,
481         std::vector<node_id_t>& node_order_vec,
482         std::vector<shortcut_edge_t>& shortcut_edge_vec)
483 {
484     CINFO_LOG( "ConstructCH::create_hierarchy starts");
485
486     int node_num = boost::num_vertices(graph);
487     Utils::init_thread_data(node_num,_contract_params.thread_num);
488
489     //split the function as two subroutines just for readablity consideration
490     this->_create_hierarchy_init(graph, shortcut_edge_vec);
491     this->_create_hierarchy_process(graph,
492             node_order_vec,
493             shortcut_edge_vec);
494
495     Utils::destroy_thread_data(_contract_params.thread_num);
496     CINFO_LOG( "ConstructCH::create_hierarchy ends");
497     return CH_RET_OK;
498 }
```

输入：一个图
输出：从前到后缩的点，添加的shortcut边
作用：根据给定图构建CH层次图

层次图构建过程中有并行操作，需要使用多线程，所以这里进行线程数据的初始化

各种数据初始化，主要是初始化了各个点的contract优先度

层次图构建主要过程

销毁线程数据

CH 层次图构建

- Utils::init(destroy)_thread_data (in 'src/ch_utils.cpp' line 1054)

```
1053 }
1054 void Utils::init_thread_data(int node_num,int thread_num){
1055     statistic_global_update_time = 0;
1056     node_pvector.resize(node_num, 0);
1057     for(int i = 0; i < thread_num; i++) {
1058         pthread_heap_ptr[i] = new ch::HeapNew<WeightHopPair>();
1059         pthread_closeTable_ptr[i] = new ch::MapNew<int>();
1060         pthread_sources_ptr[i] = new ch::MapNew<edge_descriptor_t>();
1061         pthread_targets_ptr[i] = new ch::MapNew<std::pair<edge_descriptor_t, int> >();
1062 
1063         pthread_heap_ptr[i]->init(node_num, MAX_FIND_SHORTCUT_HEAP_SIZE);
1064         pthread_closeTable_ptr[i]->init(node_num, MAX_FIND_SHORTCUT_CLOSE_TABLE_SIZE);
1065         pthread_sources_ptr[i]->init(node_num, MAX_FIND_SHORTCUT_SOURCES_SIZE);
1066         pthread_targets_ptr[i]->init(node_num, MAX_FIND_SHORTCUT_TARGETS_SIZE);
1067     }
1068 }
1069 void Utils::destroy_thread_data(int thread_num) {
1070     std::vector<double>().swap(Utils::node_pvector);
1071     for(int i = 0; i < thread_num; i++) {
1072         delete pthread_heap_ptr[i];
1073         delete pthread_closeTable_ptr[i];
1074         delete pthread_sources_ptr[i];
1075         delete pthread_targets_ptr[i];
1076     }
1077     CINFO_LOG("total global update time:%u",Utils::statistic_global_update_time);
1078 }
```

输入：图中的点数，线程数
作用：初始化每个线程的数据结构

初始化global update的累计计时
初始化每个点的优先度

初始化线程数据结构
搜索的优先队列
固定的点集合
所有起点集合
所有终点集合

输入：线程数
作用：删除每个线程的数据结构

搜索的优先队列
固定的点集合
所有起点集合
所有终点集合

MapNew是变种的map，在有效的<key,value>数目超过一定值空间就会翻倍，类似vector的实现

CH 层次图构建

- create_hierarchy (in 'src/ch_construct_ch.cpp' line 480)

```
480 ch_ret_t ConstructCH::create_hierarchy(CHGraph& graph,
481         std::vector<node_id_t>& node_order_vec,
482         std::vector<shortcut_edge_t>& shortcut_edge_vec)
483 {
484     CINFO_LOG( "ConstructCH::create_hierarchy starts");
485
486     int node_num = boost::num_vertices(graph);
487     Utils::init_thread_data(node_num,_contract_params.thread_num);
488
489     //split the function as two subroutines just for readablity consideration
490     this->_create_hierarchy_init(graph, shortcut_edge_vec);
491     this->_create_hierarchy_process(graph,
492             node_order_vec,
493             shortcut_edge_vec);
494
495     Utils::destroy_thread_data(_contract_params.thread_num);
496     CINFO_LOG( "ConstructCH::create_hierarchy ends");
497     return CH_RET_OK;
498 }
```

输入：一个图
输出：从前到后缩的点，添加的shortcut边
作用：根据给定图构建CH层次图

层次图构建过程中有并行操作，需要使用多线程，所以这里进行线程数据的初始化

各种数据初始化，主要是初始化了各个点的contract优先度

层次图构建主要过程

销毁线程数据

CH 层次图构建

- 另一个世界的故事… …

CH 层次图构建

- create_hierarchy_with_node_order (in 'src/ch_construct_ch.cpp' line 514)

```
513
514 ch_ret_t ConstructCH::create_hierarchy_with_node_order(CHGraph& graph,
515         const std::vector<node_id_t>& node_order_vec,
516         std::vector<shortcut_edge_t>& all_shortcut_edge_vec)
517 {
518     CINFO_LOG( "ConstructCH::create_hierarchy_with_ordering starts");
519
520     int num_vertex = boost::num_vertices(graph);
521     int num_edge = boost::num_edges(graph);
522
523     Utils::init_thread_data(num_vertex,1);
524
525     this->_create_hierarchy_with_node_order_init(graph, all_shortcut_edge_vec);
526
527
528     weight_map_t weight_map = boost::get(boost::edge_weight, graph);
529     shortcut_flag_map_t shortcut_flag_map = boost::get(boost::edge_shortcut_flag, graph);
530     node_order_map_t node_order_map = boost::get(boost::vertex_order, graph);
531     edge_id_map_t edge_id_map = boost::get(boost::edge_id, graph);
532
533     EdgeIDAllocator edge_id_allocator(num_edge);
534
```

输入：图，按某个给定顺序排列的点
输出：所有需要添加的shortcut
作用：在给定contract顺序下构建层次图

初始化各个线程的数据结构

初始化一些数据结构，
主要是已经被删除点的flag数组

边权“数组”（boost::property_map）
是不是shortcut“数组”
第几个被contract“数组”
每条边的id（注意shortcut的id需要新分配）

```
    for (size_t i = 0; i < node_order_vec.size(); ++i) {
        // type conversion
        node_descriptor_t node = (node_descriptor_t)node_order_vec[i];
        node_order_map[node] = i;
        //contract the node
        std::vector<shortcut_info_t> shortcut_vec;
        Utils::find_shortcuts(node, graph, _node_deleted_flag_vec, 5, shortcut_vec);

        // remove the edge and add shortcut
        _node_deleted_flag_vec[node] = true;

        // add shortcut
        for (size_t j = 0; j < shortcut_vec.size(); ++j) {

            edge_descriptor_t first_edge = shortcut_vec[j].first_edge;
            node_descriptor_t start_node = boost::source(first_edge, graph);

            edge_descriptor_t second_edge = shortcut_vec[j].second_edge;
            node_descriptor_t end_node = boost::target(second_edge, graph);

            // set edge ids
            shortcut_edge_t shortcut;
            shortcut.first_edge_id = edge_id_map[first_edge];
            shortcut.second_edge_id = edge_id_map[second_edge];
            shortcut.id = edge_id_allocator.get_next_id();
            all_shortcut_edge_vec.push_back(shortcut);

            edge_descriptor_t shortcut_edge;
            bool flag;
            boost::tie(shortcut_edge, flag) = boost::add_edge(start_node, end_node, graph);
            if (flag == false) {
                CINFO_LOG("add_edge error!,i=%u,node=%u,edge(%u,%u)",
                    i,node,start_node,end_node);
            }
            weight_map[shortcut_edge] = shortcut_vec[j].weight;
            shortcut_flag_map[shortcut_edge] = true;
            edge_id_map[shortcut_edge] = shortcut.id;

            //com_writelog(COMLOG_DEBUG, "add shortcut [start_node:%u] [end_node:%u]", start_node, end_node);
        }

        // record the shortcus
        //all_shortcut_edge_vec.insert(all_shortcut_edge_vec.end(), shortcut_vec.begin(), shortcut_vec.end());
        if (i % _contract_params.debug_trace_cycle == 0) {
            CINFO_LOG( "%d nodes are contraced, %d shortcuts added", i, all_shortcut_edge_vec.size());
        }
    }

    CINFO_LOG( "%d shortcuts added", all_shortcut_edge_vec.size());
    CINFO_LOG( "ConstructCH::create_hierarchy_with_ordering ends");

    Utils::destroy_thread_data(1);

    return CH_RET_OK;
}
```

Contract
主过程
按顺序缩点

witness path
局部搜索

删除该点

比如shortcut是v->u->w

设置一堆id属性：
v->u的id是啥
u->w的id是啥
v->w的id是啥

真正地把shortcut都加到图里面
（刚算出来的时候只是存在数组里）

设权重/shortcut标记/新添shortcut v->w的id

CH 层次图构建

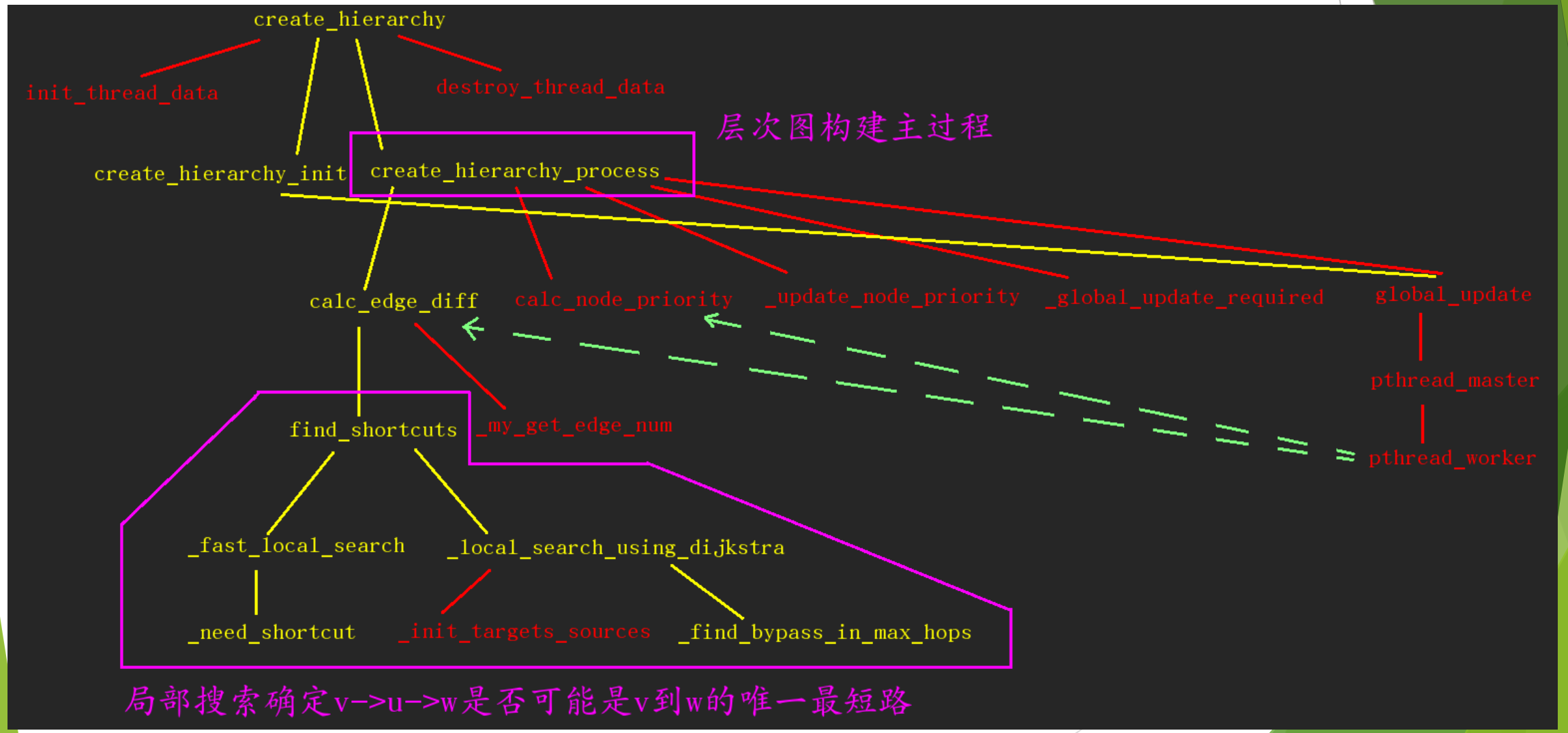
- create_hierarchy_with_node_order_init (in 'src/ch_construct_ch.cpp' line 500)

```
500 ch_ret_t ConstructCH::_create_hierarchy_with_node_order_init(CHGraph& graph,
501         std::vector<shortcut_edge_t>& shortcut_vec)
502 {
503     CINFO_LOG( "ConstructCH::create_hierarchy_with_ordering_init starts");
504     shortcut_vec.reserve(MAX_SHORTCUT_NUM);
505 
506     //init
507     int node_num = boost::num_vertices(graph);
508     _node_deleted_flag_vec.resize(node_num, false);
509     // Utils::init_static_data(node_num);
510 
511     return CH_RET_OK;
512 }
```

输入：图，待初始化的shortcut数组
作用：初始化shortcut数组和
被删除点flag数组

CH 层次图构建

- 总流程回顾



CH 查询最短路

CH 最短路查询

- 代码 Overview (ch-compiler/src)

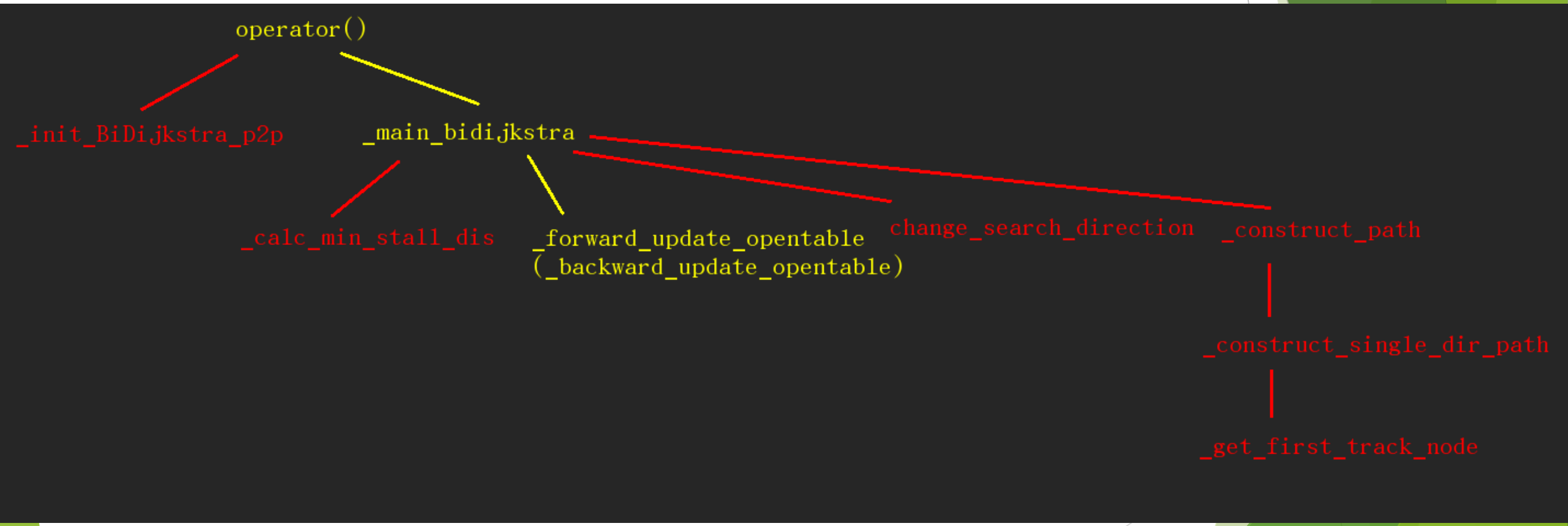
```
ch_adjgraph_types.h  ch_csrgraph_types.h     ch_dijkstra.h        ch_reader.h
ch_basetypes.h       ch_data_controller.cpp  ch_graph_builder.cpp ch_utils.cpp
ch_bfs_alt.cpp       ch_data_controller.h    ch_graph_builder.h   ch_utils.h
ch_bfs_alt.h         ch_dataunit.cpp         ch_heap.hpp          console
ch_construct_ch.cpp  ch_dataunit.h           ch_map.hpp
ch_construct_ch.h    ch_dijkstra.cpp         ch_reader.cpp
```

CH 最短路查询

- 主要函数
  - src/ch_dijkstra.cpp

CH 最短路查询

- 总流程


operator()
_init_BiDijkstra_p2p
_main_bidijkstra
_calc_min_stall_dis
_forward_update_opentable
(_backward_update_opentable)
change_search_direction
_construct_path
_construct_single_dir_path
_get_first_track_node

CH 最短路查询

- CHDijkstra::operator() (in 'src/ch_dijkstra.cpp' line 500)

```
482 ret_t CHDijkstra::operator()(node_descriptor_t src_node,
483         node_descriptor_t dst_node,
484         const CHGraph& ch_graph,
485         weight_t& min_weight,
486         std::vector<edge_descriptor_t>& path_edge_vec)
487 {
488
489     _init_BiDijkstra_p2p(src_node,dst_node);
490
491     return _main_bidijkstra(ch_graph,min_weight,path_edge_vec);
492
493 }
```

输入：起点，终点，层次图
输出：最短路长度，最短路
作用：通过双向dijkstra得到起点到终点的最短路

初始化双向dijkstra需要的数据结构（主要是各种优先队列）

双向dijkstra找最短路

CH 最短路查询

- CHDijkstra::_init_BiDijkstra_p2p (in 'src/ch_dijkstra.cpp' line 236)

```
236 void CHDijkstra::_init_BiDijkstra_p2p(node_descriptor_t src_node,node_descriptor_t dst_node)
237 {
238     _search_direction = FORWARD;
239     _reverse_direction = BACKWARD;
240
241     _open_table[FORWARD].clear();
242     _open_table[BACKWARD].clear();
243
244     _close_table[FORWARD].clear();
245     _close_table[BACKWARD].clear();
246
247     _stall_table[FORWARD].clear();
248     _stall_table[BACKWARD].clear();
249
250     open_node_t init_open_node;
251
252     //init forward search open table
253     init_open_node.prev_node_id = src_node;
254     init_open_node.cur_node_id = src_node;
255     init_open_node.weight = 0;
256     _open_table[_search_direction].insert(init_open_node);
257
258     //init backward search open table
259     init_open_node.prev_node_id = dst_node;
260     init_open_node.cur_node_id = dst_node;
261     init_open_node.weight = 0;
262     _open_table[_reverse_direction].insert(init_open_node);
263
264     _weight = std::numeric_limits<weight_t>::max();
265     _meeting_u = std::numeric_limits<node_id_t>::max();
266     _meeting_v = std::numeric_limits<node_id_t>::max();
267 }
```

输入：起点，终点
作用：初始化数据结构

清空
所有待扩展点的表（open表）
所有已经固定点的表（close表）
所有被stall-on-demand优化暂停的点的表（stall表）

初始化正向的open表，只有起点

初始化逆向的open表，只有终点

初始化当前最短路径长度为无穷
初始化正/逆方向相遇点

CH 最短路查询

- CHDijkstra::operator() (in 'src/ch_dijkstra.cpp' line 500)

```
482 ret_t CHDijkstra::operator()(node_descriptor_t src_node,
483         node_descriptor_t dst_node,
484         const CHGraph& ch_graph,
485         weight_t& min_weight,
486         std::vector<edge_descriptor_t>& path_edge_vec)
487 {
488
489     _init_BiDijkstra_p2p(src_node,dst_node);
490
491     return _main_bidijkstra(ch_graph,min_weight,path_edge_vec);
492
493 }
```

输入：起点，终点，层次图
输出：最短路长度，最短路
作用：通过双向dijkstra得到起点到终点的最短路

初始化双向dijkstra需要的数据结构（主要是各种优先队列）

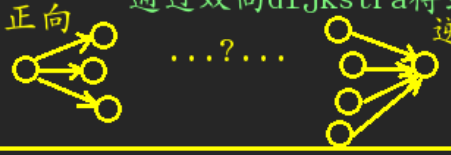
双向dijkstra找最短路

CH 最短路查询

- CHDijkstra::_main_bidijkstra (in 'src/ch_dijkstra.cpp' line 398)

```
394         }
395     }
396 }
397
398 ret_t CHDijkstra::_main_bidijkstra(const CHGraph& ch_graph,
399     weight_t& min_weight,
400     std::vector<edge_descriptor_t>& path_edge_vec)
401 {
402     // main biDijkstra algorithm
403     ret_t ret = RET_ERROR;
404
405     while (1)
406     {
407         //const edge_id_map_t& edge_id_map = boost::get(boost::edge_id, const_cast<CHGraph&>(ch_graph));
408
409         const open_node_t cur_node = _open_table[_search_direction].top();
410         _open_table[_search_direction].pop();
411
412         node_descriptor_t cur_vertex = cur_node.cur_node_id;
413
414         //try stall node
415         weight_t min_stall_dis = _calc_min_stall_dis(cur_vertex,ch_graph);
416         if (min_stall_dis < cur_node.weight) {
417             stall_node_t stall_node;
418             stall_node.weight = min_stall_dis;
419             _stall_table[_search_direction].insert(cur_node.cur_node_id, stall_node);
420         } else {
421             close_node_t close_node;
422
423             close_node.prev_node_id = cur_node.prev_node_id;
424             close_node.weight = cur_node.weight;
425             close_node.ed = cur_node.ed;
426             _close_table[_search_direction].insert(cur_node.cur_node_id, close_node);
427             /*CDEBUG_LOG("search direction: %u, close table node id:%u",
428                     _search_direction,
429                     cur_node.cur_node_id);*/
430
431             //update open table
432             if (_search_direction == FORWARD) {
433                 _forward_update_opentable(cur_node,cur_vertex,ch_graph);
434             } else {
435                 _backward_update_opentable(cur_node,cur_vertex,ch_graph);
436             }
437         }
438
439         bool is_search_stop  = _open_table[ _search_direction].size()==0 || _open_table[ _search_direction].top().weight>=_weight;
440         bool is_reverse_stop = _open_table[_reverse_direction].size()==0 || _open_table[_reverse_direction].top().weight>=_weight;
441
442         if(false == is_reverse_stop) {
443             change_search_direction();
444         } else if(true == is_search_stop) {
445             break;
446         }
447     }
```

输入：层次图
输出：最短路长度，最短路
作用：根据当前open表里的点
通过双向dijkstra得到最短路



双向Dijkstra找最短路

得到open表（可扩展节点表）中的最小值节点

Stall-on-demand优化

如果起点到当前点不是最优路径，那么Stall当前点，将其加入Stall表

否则，将其固定（加入close表），进行扩展

更改搜索方向（正向->逆向；逆向->正向）

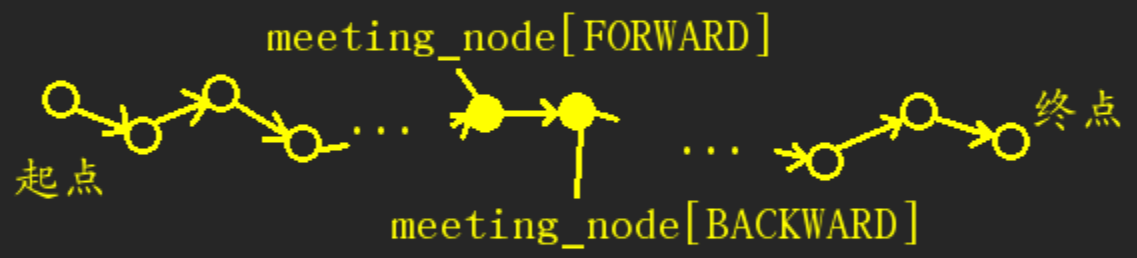
当open表为空或起点到open表中的点的最小距离不小于最短路长度时，结束

实际上，结束条件可以更弱

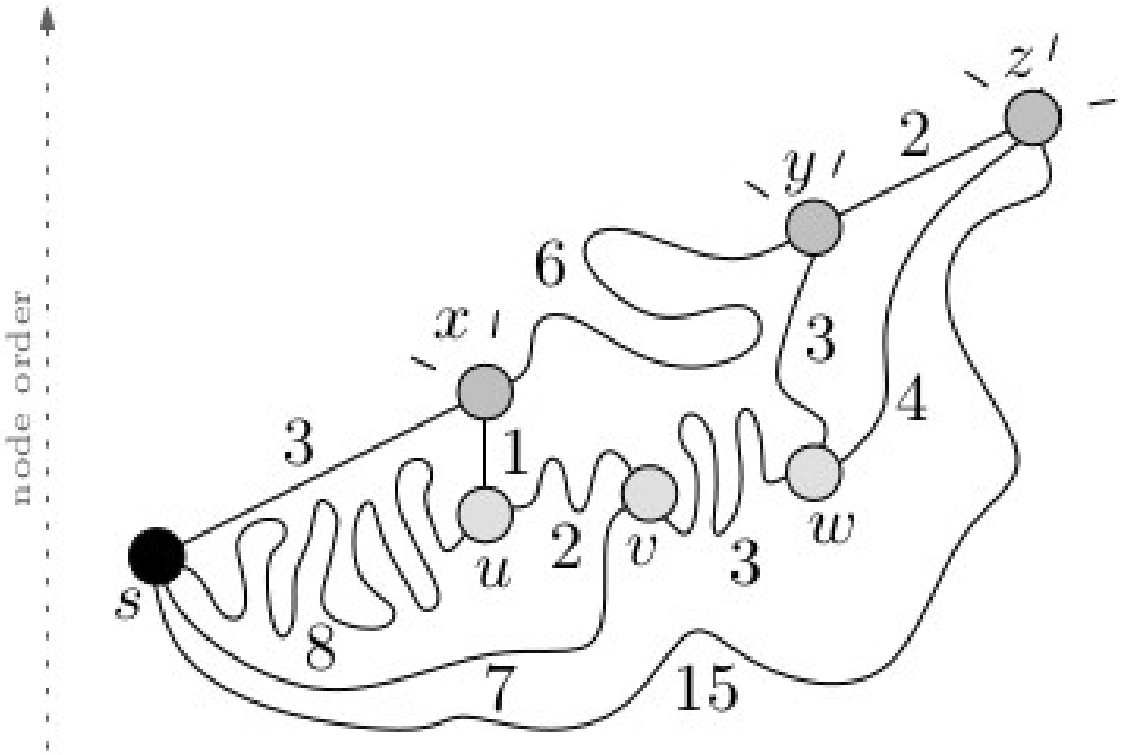
```
449     if(_weight != std::numeric_limits<weight_t>::max()) {
450         ret = RET_OK;
451     } else {
452         CDEBUG_LOG("src and dst is not connected in the road network");
453         return RET_NOT_CONNECTED;
454     }
455     min_weight = _weight;
456
457     CDEBUG_LOG("[min_weight:%u]", min_weight);
458
459     //generate path from src_node to dst_node
460     node_id_t meeting_nodes[2];
461
462     if (_meeting_direction == FORWARD) {
463         meeting_nodes[FORWARD] = _meeting_u;
464         meeting_nodes[BACKWARD] = _meeting_v;
465     }
466     else {
467         meeting_nodes[FORWARD] = _meeting_v;
468         meeting_nodes[BACKWARD] = _meeting_u;
469     }
470
471     ret = _construct_path(_open_table,
472             _close_table,
473             meeting_nodes,
474             _meeting_direction,
475             path_edge_vec);
476
477     RET_CHECK_ERROR(ret);
478
479     return ret;
480 }
```

最短路径长度为无穷，
表明起点到终点没有路径

更新最短路长度



还原最短路径


node order
s
u
v
w
x′
y′
z′
3
1
2
3
6
3
4
2
8
7
15

CH 最短路查询

- CHDijkstra::_calc_min_stall_dis (in 'src/ch_dijkstra.cpp' line 165)

```cpp
weight_t CHDijkstra::_calc_min_stall_dis(node_descriptor_t cur_vertex,
    const CHGraph& ch_graph)
{
    weight_t min_stall_dis = std::numeric_limits<weight_t>::max();
    if ( search_direction == FORWARD) {
        BOOST_FOREACH( edge_descriptor_t ed, boost::in_edges(cur_vertex, ch_graph) ) {
            if (ch_graph[ed].weight >= MAX_EDGE_WEIGHT) {
                continue;
            }
            node_descriptor_t adj_vertex;
            adj_vertex = boost::source(ed, ch_graph);
            if (ch_graph[adj_vertex].order < ch_graph[cur_vertex].order) {
                continue;
            }
            open_node_t search_open_node;
            close_node_t search_close_node;
            stall_node_t search_stall_node;
            if (_close_table[_search_direction].find(adj_vertex, search_close_node)) {
                weight_t dis = search_close_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
            else if (_stall_table[_search_direction].find(adj_vertex, search_stall_node)) {
                weight_t dis = search_stall_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
            else if (_open_table[_search_direction].find(adj_vertex, search_open_node)) {
                weight_t dis = search_open_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
        }
    } else { // search direction == FORWARD
        BOOST_FOREACH( edge_descriptor_t ed, boost::out_edges(cur_vertex, ch_graph) ) {
            if (ch_graph[ed].weight >= MAX_EDGE_WEIGHT) {
                continue;
            }
            node_descriptor_t adj_vertex;
            adj_vertex = boost::target(ed, ch_graph);
            if (ch_graph[adj_vertex].order < ch_graph[cur_vertex].order) {
                continue;
            }
            open_node_t search_open_node;
            close_node_t search_close_node;
            stall_node_t search_stall_node;
            if (_close_table[_search_direction].find(adj_vertex, search_close_node)) {
                weight_t dis = search_close_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
            else if (_stall_table[_search_direction].find(adj_vertex, search_stall_node)) {
                weight_t dis = search_stall_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
            else if (_open_table[_search_direction].find(adj_vertex, search_open_node)) {
                weight_t dis = search_open_node.weight + ch_graph[ed].weight;
                if(dis < min_stall_dis) {
                    min_stall_dis = dis;
                }
            }
        }
    }
    return min_stall_dis;
}
```

输入：某个点u，层次图
输出：用来stall点u的某路径s->...->u的长度
作用：找一条尽量短的路径stall点u

必须比点u高

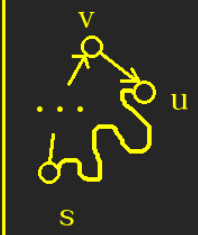
看看v是否属于三个表中的一个

翻翻close表

翻翻stall表

翻翻open表

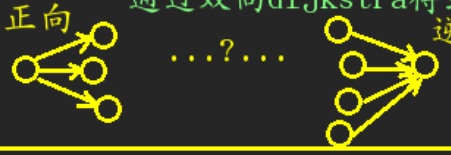
正向搜索
枚举比点u高的
点'v'->u尝试



逆向搜索情况
类似，不过枚
举的是出度点
u->'w'

```
394         }
395     }
396 }
397
398 ret_t CHDijkstra::_main_bidijkstra(const CHGraph& ch_graph,
399     weight_t& min_weight,
400     std::vector<edge_descriptor_t>& path_edge_vec)
401 {
402     // main biDijkstra algorithm
403     ret_t ret = RET_ERROR;
404
405     while (1)
406     {
407         //const edge_id_map_t& edge_id_map = boost::get(boost::edge_id, const_cast<CHGraph&>(ch_graph));
408
409         const open_node_t cur_node = _open_table[_search_direction].top();
410         _open_table[_search_direction].pop();
411
412         node_descriptor_t cur_vertex = cur_node.cur_node_id;
413
414         //try stall node
415         weight_t min_stall_dis = _calc_min_stall_dis(cur_vertex,ch_graph);
416         if (min_stall_dis < cur_node.weight) {
417             stall_node_t stall_node;
418             stall_node.weight = min_stall_dis;
419             _stall_table[_search_direction].insert(cur_node.cur_node_id, stall_node);
420         } else {
421             close_node_t close_node;
422
423             close_node.prev_node_id = cur_node.prev_node_id;
424             close_node.weight = cur_node.weight;
425             close_node.ed = cur_node.ed;
426             _close_table[_search_direction].insert(cur_node.cur_node_id, close_node);
427             /*CDEBUG_LOG("search direction: %u, close table node id:%u",
428                     _search_direction,
429                     cur_node.cur_node_id);*/
430
431             //update open table
432             if (_search_direction == FORWARD) {
433                 _forward_update_opentable(cur_node,cur_vertex,ch_graph);
434             } else {
435                 _backward_update_opentable(cur_node,cur_vertex,ch_graph);
436             }
437         }
438
439         bool is_search_stop  = _open_table[ _search_direction].size()==0 || _open_table[ _search_direction].top().weight>=_weight;
440         bool is_reverse_stop = _open_table[_reverse_direction].size()==0 || _open_table[_reverse_direction].top().weight>=_weight;
441
442         if(false == is_reverse_stop) {
443             change_search_direction();
444         } else if(true == is_search_stop) {
445             break;
446         }
447     }
```

输入：层次图
输出：最短路长度，最短路
作用：根据当前open表里的点
通过双向dijkstra得到最短路



双向Dijkstra找最短路

得到open表（可扩展节点表）中的最小值节点

Stall-on-demand优化

如果起点到当前点不是最优路径，那么Stall当前点，将其加入Stall表

否则，将其固定（加入close表），进行扩展

更改搜索方向（正向->逆向；逆向->正向）

当open表为空或起点到open表中的点的最小距离不小于最短路长度时，结束

实际上，结束条件可以更弱

CH 最短路查询

- CHDijkstra::_forward_update_opentable (in 'src/ch_dijkstra.cpp' line 268)

```
266     _meeting_v = std::numeric_limits<node_id_t>::max();
267 }
268 void CHDijkstra::_forward_update_opentable(const open_node_t cur_node,
269     node_descriptor_t cur_vertex,
270     const CHGraph& ch_graph)
271 {
272     BOOST_FOREACH( edge_descriptor_t ed, boost::out_edges(cur_vertex, ch_graph) ) {
273         if (ch_graph[ed].weight >= MAX_EDGE_WEIGHT) {
274             continue;
275         }
276         node_descriptor_t adj_vertex;
277         adj_vertex = boost::target(ed, ch_graph);
278 
279         //only use upward subgraph
280         if (ch_graph[adj_vertex].order < ch_graph[cur_vertex].order) {
281             continue;
282         }
283 
284         node_id_t next_node_id = adj_vertex;
285 
286         weight_t next_node_weight = cur_node.weight + ch_graph[ed].weight;
287 
288         open_node_t search_open_node;
289         close_node_t search_close_node;
290         stall_node_t search_stall_node;
291         if (_close_table[_search_direction].find(next_node_id, search_close_node)) {
292             //DO NOTHING
293         } else if (_stall_table[_search_direction].find(next_node_id, search_stall_node)) {
294             //DO NOTHING
295         } else if (_open_table[_search_direction].find(next_node_id, search_open_node)) {
296             if (next_node_weight < search_open_node.weight) {
297                 search_open_node.prev_node_id = cur_node.cur_node_id;
298                 search_open_node.weight = next_node_weight;
299                 search_open_node.ed = ed;
300                 _open_table[_search_direction].update(search_open_node);
301             }
302         } else {
303             search_open_node.prev_node_id = cur_node.cur_node_id;
304             search_open_node.cur_node_id = next_node_id;
305             search_open_node.weight = next_node_weight;
306             search_open_node.ed = ed;
307             _open_table[_search_direction].insert(search_open_node);
308         }
309 
310         open_node_t reverse_open_node;
311         close_node_t reverse_close_node;
312         bool reverse_open_meeting =
313             _open_table[_reverse_direction].find(next_node_id, reverse_open_node);
314         bool reverse_close_meeting =
315             _close_table[_reverse_direction].find(next_node_id, reverse_close_node);
316         if (reverse_open_meeting == true || reverse_close_meeting == true) {
317             weight_t dis = 0;
318             if (reverse_open_meeting == true) {
319                 dis = next_node_weight + reverse_open_node.weight;
320             } else {
321                 dis = next_node_weight + reverse_close_node.weight;
322             }
323             if (dis < _weight) {
324                 _weight = dis;
325                 _meeting_u = cur_node.cur_node_id;
326                 _meeting_ed = ed;
327                 _meeting_v = next_node_id;
328                 _meeting_direction = _search_direction;
329             }
330         }
331     }
332 }
```

输入：当前点u，
当前点u的node descriptor，
层次图
作用：在双向dijkstra中正向扩展点u

枚举u->'w'

有效边

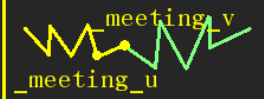
点w层次在u之上（往上走）

close/stall表中的点不可能再被更新了（根据dijkstra算法的性质）

如果在open表中，则更新

否则，加入open表

如果逆向搜索也已经搜索到该点，则得到一条路径



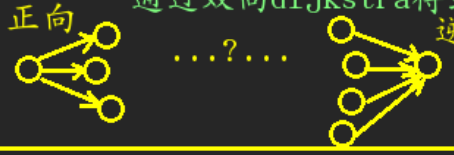
更新最短路径长度和两个方向相遇的点

CH 最短路查询

- CHDijkstra::_backward_update_opentable (in 'src/ch_dijkstra.cpp' line 333)
- 类似，不过枚举方向为 'v'->u ，其他相同

```
394         }
395     }
396 }
397
398 ret_t CHDijkstra::_main_bidijkstra(const CHGraph& ch_graph,
399     weight_t& min_weight,
400     std::vector<edge_descriptor_t>& path_edge_vec)
401 {
402     // main biDijkstra algorithm
403     ret_t ret = RET_ERROR;
404
405     while (1)
406     {
407         //const edge_id_map_t& edge_id_map = boost::get(boost::edge_id, const_cast<CHGraph&>(ch_graph));
408
409         const open_node_t cur_node = _open_table[_search_direction].top();
410         _open_table[_search_direction].pop();
411
412         node_descriptor_t cur_vertex = cur_node.cur_node_id;
413
414         //try stall node
415         weight_t min_stall_dis = _calc_min_stall_dis(cur_vertex,ch_graph);
416         if (min_stall_dis < cur_node.weight) {
417             stall_node_t stall_node;
418             stall_node.weight = min_stall_dis;
419             _stall_table[_search_direction].insert(cur_node.cur_node_id, stall_node);
420         } else {
421             close_node_t close_node;
422
423             close_node.prev_node_id = cur_node.prev_node_id;
424             close_node.weight = cur_node.weight;
425             close_node.ed = cur_node.ed;
426             _close_table[_search_direction].insert(cur_node.cur_node_id, close_node);
427             /*CDEBUG_LOG("search direction: %u, close table node id:%u",
428                     _search_direction,
429                     cur_node.cur_node_id);*/
430
431             //update open table
432             if (_search_direction == FORWARD) {
433                 _forward_update_opentable(cur_node,cur_vertex,ch_graph);
434             } else {
435                 _backward_update_opentable(cur_node,cur_vertex,ch_graph);
436             }
437         }
438
439         bool is_search_stop  = _open_table[ _search_direction].size()==0 || _open_table[ _search_direction].top().weight>=_weight;
440         bool is_reverse_stop = _open_table[_reverse_direction].size()==0 || _open_table[_reverse_direction].top().weight>=_weight;
441
442         if(false == is_reverse_stop) {
443             change_search_direction();
444         } else if(true == is_search_stop) {
445             break;
446         }
447     }
```

输入：层次图
输出：最短路长度，最短路
作用：根据当前open表里的点
通过双向dijkstra得到最短路



双向Dijkstra找最短路

得到open表（可扩展节点表）中的最小值节点

Stall-on-demand优化

如果起点到当前点不是最优路径，那么Stall当前点，将其加入Stall表

否则，将其固定（加入close表），进行扩展

更改搜索方向（正向->逆向；逆向->正向）

当open表为空或起点到open表中的点的最小距离不小于最短路长度时，结束

实际上，结束条件可以更弱

CH 最短路查询

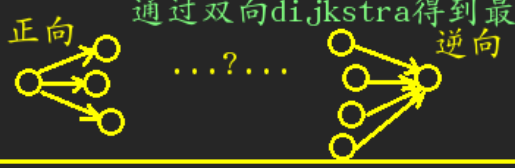
- CHDijkstra::change_search_direction (in 'src/ch_dijkstra.cpp' line 633)

```
633 inline void CHDijkstra::change_search_direction()
634 {
635     std::swap(_search_direction, _reverse_direction);
636 }
```

作用：改变双向dijkstra的搜索方向

```
394         }
395     }
396 }
397
398 ret_t CHDijkstra::_main_bidijkstra(const CHGraph& ch_graph,
399     weight_t& min_weight,
400     std::vector<edge_descriptor_t>& path_edge_vec)
401 {
402     // main biDijkstra algorithm
403     ret_t ret = RET_ERROR;
404
405     while (1)
406     {
407         //const edge_id_map_t& edge_id_map = boost::get(boost::edge_id, const_cast<CHGraph&>(ch_graph));
408
409         const open_node_t cur_node = _open_table[_search_direction].top();
410         _open_table[_search_direction].pop();
411
412         node_descriptor_t cur_vertex = cur_node.cur_node_id;
413
414         //try stall node
415         weight_t min_stall_dis = _calc_min_stall_dis(cur_vertex,ch_graph);
416         if (min_stall_dis < cur_node.weight) {
417             stall_node_t stall_node;
418             stall_node.weight = min_stall_dis;
419             _stall_table[_search_direction].insert(cur_node.cur_node_id, stall_node);
420         } else {
421             close_node_t close_node;
422
423             close_node.prev_node_id = cur_node.prev_node_id;
424             close_node.weight = cur_node.weight;
425             close_node.ed = cur_node.ed;
426             _close_table[_search_direction].insert(cur_node.cur_node_id, close_node);
427             /*CDEBUG_LOG("search direction: %u, close table node id:%u",
428                     _search_direction,
429                     cur_node.cur_node_id);*/
430
431             //update open table
432             if (_search_direction == FORWARD) {
433                 _forward_update_opentable(cur_node,cur_vertex,ch_graph);
434             } else {
435                 _backward_update_opentable(cur_node,cur_vertex,ch_graph);
436             }
437         }
438
439         bool is_search_stop  = _open_table[ _search_direction].size()==0 || _open_table[ _search_direction].top().weight>=_weight;
440         bool is_reverse_stop = _open_table[_reverse_direction].size()==0 || _open_table[_reverse_direction].top().weight>=_weight;
441
442         if(false == is_reverse_stop) {
443             change_search_direction();
444         } else if(true == is_search_stop) {
445             break;
446         }
447     }
```

输入：层次图
输出：最短路长度，最短路
作用：根据当前open表里的点
通过双向dijkstra得到最短路



双向Dijkstra找最短路

得到open表（可扩展节点表）中的最小值节点

Stall-on-demand优化

如果起点到当前点不是最优路径，那么Stall当前点，将其加入Stall表

否则，将其固定（加入close表），进行扩展

更改搜索方向（正向->逆向；逆向->正向）

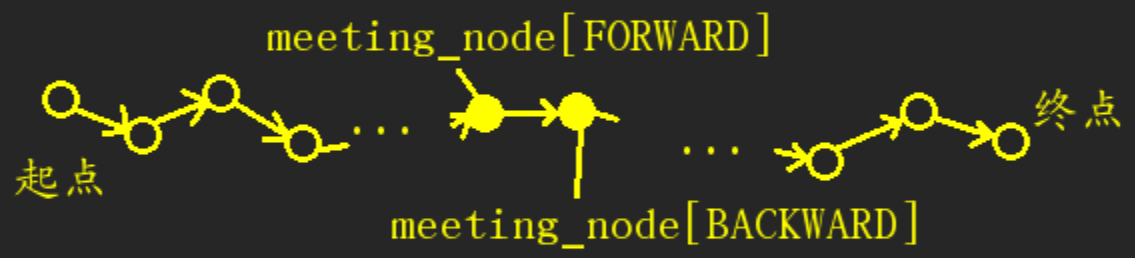
当open表为空或起点到open表中的点的最小距离不小于最短路长度时，结束

实际上，结束条件可以更弱

```
449     if(_weight != std::numeric_limits<weight_t>::max()) {
450         ret = RET_OK;
451     } else {
452         CDEBUG_LOG("src and dst is not connected in the road network");
453         return RET_NOT_CONNECTED;
454     }
455     min_weight = _weight;
456
457     CDEBUG_LOG("[min_weight:%u]", min_weight);
458
459     //generate path from src_node to dst_node
460     node_id_t meeting_nodes[2];
461
462     if (_meeting_direction == FORWARD) {
463         meeting_nodes[FORWARD] = _meeting_u;
464         meeting_nodes[BACKWARD] = _meeting_v;
465     }
466     else {
467         meeting_nodes[FORWARD] = _meeting_v;
468         meeting_nodes[BACKWARD] = _meeting_u;
469     }
470
472     ret = _construct_path(_open_table,
473             _close_table,
474             meeting_nodes,
475             _meeting_direction,
476             path_edge_vec);
477     RET_CHECK_ERROR(ret);
478
479     return ret;
480 }
```

最短路径长度为无穷，
表明起点到终点没有路径

更新最短路长度



还原最短路径

CH 最短路查询

- CHDijkstra::_construct_path (in 'src/ch_dijkstra.cpp' line 588)

```
587
588 ret_t CHDijkstra::_construct_path(const OpenTable open_tables[],
589         const CloseTable close_tables[],
590         const node_id_t meeting_nodes[],
591         search_direction_t meeting_direction,
592         std::vector<edge_descriptor_t>& path_edge_vec)
593 {
594     /// □ □ □ □ □ □ □ □
595     std::vector<edge_descriptor_t> forward_path_edge_vec;
596     ret_t ret = _construct_single_dir_path(open_tables[FORWARD],
597             close_tables[FORWARD],
598             meeting_nodes[FORWARD],
599             forward_path_edge_vec);
600     if (ret != RET_OK) {
601         CWARNING_LOG("%s fails", __PRETTY_FUNCTION__);
602         return RET_ERROR;
603     }
604
605     /// the forward path is reversed, so reverse it back
606     std::reverse(forward_path_edge_vec.begin(), forward_path_edge_vec.end());
607
608
609
610     /// □ □ □ □ □ □ □ □
611     std::vector<edge_descriptor_t> backward_path_edge_vec;
612     ret = _construct_single_dir_path(open_tables[BACKWARD],
613             close_tables[BACKWARD],
614             meeting_nodes[BACKWARD],
615             backward_path_edge_vec);
616     if (ret != RET_OK) {
617         CWARNING_LOG("%s fails", __PRETTY_FUNCTION__);
618         return RET_ERROR;
619     }
620
621     /// □ □ □ □
622     path_edge_vec.insert(path_edge_vec.end(), forward_path_edge_vec.begin(), forward_path_edge_vec.end());
623     /// □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ edge
624     path_edge_vec.push_back(_meeting_ed);
625     path_edge_vec.insert(path_edge_vec.end(), backward_path_edge_vec.begin(), backward_path_edge_vec.end());
626
627     CDEBUG_LOG("forward_path_edge_vec size:%d", forward_path_edge_vec.size());
628     CDEBUG_LOG("backward_path_edge_vec size:%d", backward_path_edge_vec.size());
629
630     return RET_OK;
631 }
```

输入：open表，
close表，
相遇的两个点，
相遇时的搜索方向
输出：最短路径
作用：通过相遇点还原最短路径

顺藤摸瓜找回起点

因为是倒着找的(u<-...<-s)，
所以要再倒过来（s->...->u）

顺藤摸瓜回到终点

把
s->...->u,
u->v,
v->...->t
3段拼起来，
就得到了
s->...->t的
最短路径

CH 最短路查询

- CHDijkstra::_construct_single_dir_path (in 'src/ch_dijkstra.cpp' line 540)

```
ret_t CHDijkstra::_construct_single_dir_path(const OpenTable& open_table,
        const CloseTable& close_table,
        const node_id_t meeting_node,
        std::vector<edge_descriptor_t>& path_edge_vec)
{
    node_id_t back_track_id = meeting_node;

    edge_descriptor_t edge;
    close_node_t back_close_track_node;
    /// this is a trick, because first node may still in open table
    {
        node_id_t pre_back_track_id;
        ret_t ret = _get_first_track_node(back_track_id,
            open_table,
            close_table,
            edge,
            pre_back_track_id);

        /// □□□□□□□□□□□□□□□□□□□□ case □□□□□□□
        if (back_track_id == pre_back_track_id) {
            CDEBUG_LOG("init node meet node_id:%u", back_track_id);
            return RET_OK;
        }

        if (ret != RET_OK) {
            CWARNING_LOG("%s fails", __PRETTY_FUNCTION__);
            return RET_ERROR;
        }
        path_edge_vec.push_back(edge);
        back_track_id = pre_back_track_id;
    }

    while (1) {
        if (close_table.find(back_track_id,
                             back_close_track_node) == false) {
            CWARNING_LOG("fail to find back_track_id:%d in close talbe", back_track_id);
            return RET_GEN_ROUTE_FAIL;
        }
        else if (back_track_id == back_close_track_node.prev_node_id) {
            break;
        }
        path_edge_vec.push_back(back_close_track_node.ed);
        back_track_id = back_close_track_node.prev_node_id;
    }

    return RET_OK;
}
```

输入：open表，
close表，
相遇两点中的一个点u
输出：起点到点u（点u到终点）的最短路径
注意：该最短路径还没有将shortcut解压，
是层次图中的最短路径

得到点u在3个表中的对应点
（如果点u是扩展的点，那么就在close表里；
如果点u是被扩展的点，那么就在open表里）

路径上只有一个点的情况

在路径中加入第一个点

剩下的点肯定在close表里，
顺藤摸瓜即可

只有起点（终点）有自环，结束

CH 最短路查询

- CHDijkstra::_get_first_track_node (in 'src/ch_dijkstra.cpp' line 638)

```
638 ret_t CHDijkstra::_get_first_track_node(const node_id_t back_track_id,
639     const OpenTable& open_table,
640     const CloseTable& close_table,
641     edge_descriptor_t& edge,
642     node_id_t& pre_back_track_id)
643 {
644     close_node_t back_close_track_node;
645     open_node_t back_open_track_node;
646 
647     if (close_table.find(back_track_id, back_close_track_node) == true) {
648         edge = back_close_track_node.ed;
649         pre_back_track_id = back_close_track_node.prev_node_id;
650         return RET_OK;
651     }
652     else if (open_table.find(back_track_id, back_open_track_node) == true) {
653         edge = back_open_track_node.ed;
654         pre_back_track_id = back_open_track_node.prev_node_id;
655         return RET_OK;
656     }
657     else {
658         CWARNING_LOG("failed to find node:%d in close_table or open_table", back_track_id);
659         return RET_ERROR;
660     }
661 }
```

输入：相遇的点u，
open表，
close表
输出：第一条回溯边
正向：u<-v，逆向：u->v
点u向前回溯一条边的点v

在open表里找点u对应的点

在close表里找点u对应的点

点u对应的点只可能在这两个表中，否则肯定有bug

CH 层次图构建

- 另一个世界的故事… …

CH 最短路查询

- CHDijkstra::operator() (in 'src/ch_dijkstra.cpp' line 530)

```
530 ret_t CHDijkstra::operator()(std::vector<node_descriptor_t> src_node_vec,
531             std::vector<node_descriptor_t> dst_node_vec,
532             const CHGraph& ch_graph,
533             weight_t& min_weight,
534             std::vector<edge_descriptor_t>& path_edge_vec)
535 {
536     _init_BiDijkstra_mp2mp(src_node_vec,dst_node_vec);
537
538     return _main_bidijkstra(ch_graph,min_weight,path_edge_vec);
539 }
```

输入：起点集合，终点集合，层次图
输出：最短路长度，最短路
作用：通过双向dijkstra得到起点集合到终点集合间最短的一条路径

初始化双向dijkstra需要的数据结构（各种优先队列）

双向dijkstra找最短路

CH 最短路查询

- CHDijkstra::_init_BiDijkstra_mp2mp (in 'src/ch_dijkstra.cpp' line 494)

```
494 void CHDijkstra::_init_BiDijkstra_mp2mp(std::vector<node_descriptor_t> src_node_vec,std::vector<node_descriptor_t> dst_node_vec)
495 {
496     _search_direction = FORWARD;
497     _reverse_direction = BACKWARD;
498
499     _open_table[FORWARD].clear();
500     _open_table[BACKWARD].clear();
501
502     _close_table[FORWARD].clear();
503     _close_table[BACKWARD].clear();
504
505     _stall_table[FORWARD].clear();
506     _stall_table[BACKWARD].clear();
507
508     open_node_t init_open_node;
509     init_open_node.weight = 0;
510
511     //init forward search open table
512     std::vector<node_descriptor_t>::iterator itr;
513     for(itr = src_node_vec.begin(); itr != src_node_vec.end(); itr++) {
514         init_open_node.prev_node_id = *itr;
515         init_open_node.cur_node_id = *itr;
516         _open_table[_search_direction].insert(init_open_node);
517     }
518
519     //init backward search open table
520     for(itr = dst_node_vec.begin(); itr != dst_node_vec.end(); itr++) {
521         init_open_node.prev_node_id = *itr;
522         init_open_node.cur_node_id = *itr;
523         _open_table[_reverse_direction].insert(init_open_node);
524     }
525
526     _weight = std::numeric_limits<weight_t>::max();
527     _meeting_u = std::numeric_limits<node_id_t>::max();
528     _meeting_v = std::numeric_limits<node_id_t>::max();
529 }
```

输入：起点集合，终点集合
作用：初始化数据结构

清空
所有待扩展点的表（open表）
所有已经固定点的表（close表）
所有被stall-on-demand优化暂停的点的表（stall表）

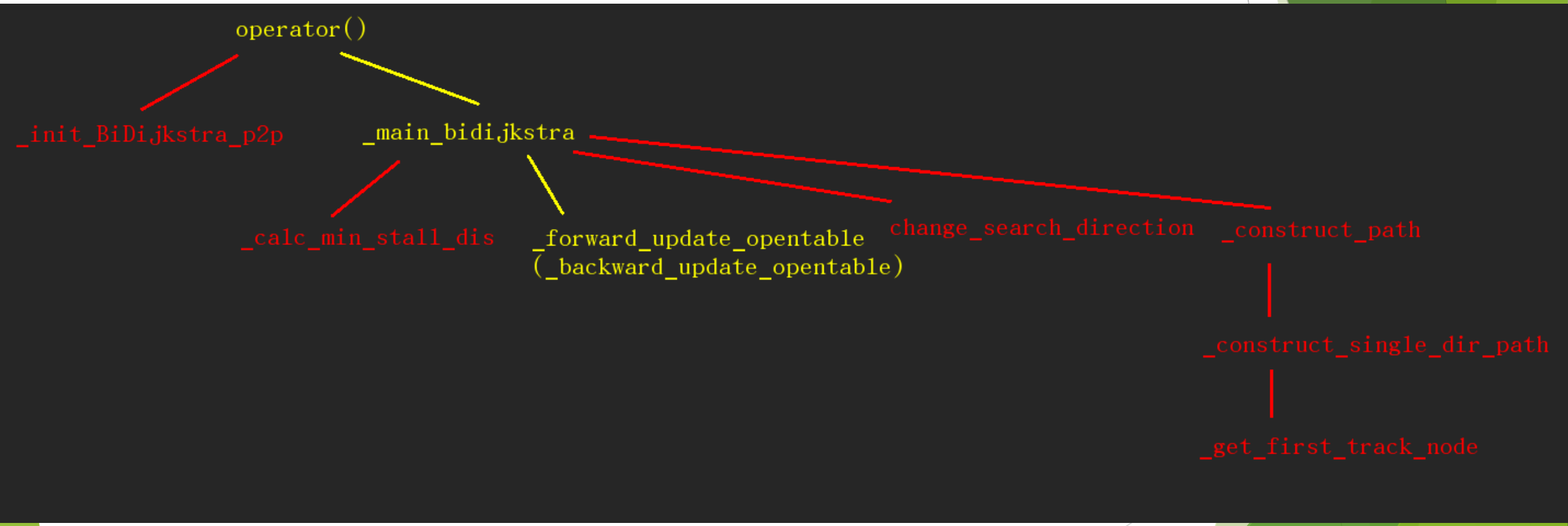
初始化正向的open表，只有起点集合

初始化逆向的open表，只有终点集合

初始化当前最短路径长度为无穷
初始化正/逆方向相遇点

CH 最短路查询

- 总流程回顾



Q&A?

CH 解压最短路

- CHDataUnit::revert_shortcut (in 'src/ch_dataunit.cpp' line 205)

```
205 void CHDataUnit::revert_shortcut(const std::vector<edge_descriptor_t>& route_ed_list,
206                     std::vector<topo_link_id_t>& route_id_vec,
207                     std::vector<ch::weight_t>& route_weight_vec) const
208 {
209     //debug
210     const CHGraph& ch_graph = this->graph;
211 
212     CDEBUG_LOG("Edge ID List");
213     for (size_t i = 0; i < route_ed_list.size(); ++i) {
214         edge_descriptor_t ed = route_ed_list[i];
215         topo_link_id_t link_id = ch_graph[ed].edge_id;
216         CDEBUG_LOG("Edge_id: %d", link_id);
217     }
218 
219     CDEBUG_LOG("Edge ID List done");
220 
221     for (size_t i = 0; i < route_ed_list.size(); ++i) {
222         // Topo □ □ shortcut
223         edge_descriptor_t ed = route_ed_list[i];
224         if (ch_graph[ed].edge_shortcut_flag == true) {
225             /*com_writelog(COMLOG_DEBUG, "Edge_id: %d, Weight : %d.", */
226                     /*ch_graph[ed].edge_id, ch_graph[ed].weight);*/
227             std::vector<topo_link_id_t> link_id_vec;
228             std::vector<ch::weight_t> weight_vec;
229             link_id_vec.reserve(1024);
230             weight_vec.reserve(1024);
231             this->revert(ed,
232                 link_id_vec,
233                 weight_vec);
234             std::copy(link_id_vec.begin(), link_id_vec.end(), std::back_inserter(route_id_vec));
235             std::copy(weight_vec.begin(), weight_vec.end(), std::back_inserter(route_weight_vec));
236         } else {
237             route_id_vec.push_back(ch_graph[ed].edge_id);
238             route_weight_vec.push_back(ch_graph[ed].weight);
239             /*com_writelog(COMLOG_DEBUG, "Edge_id: %d, Weight : %d.", */
240                     /*ch_graph[ed].edge_id, ch_graph[ed].weight);*/
241 
242         }
243     }
244 }
245 
```

输入：层次图中的最短路
输出：原图中的最短路（边id和边权

输出打印信息

解压层次图最短路上的shortcut

是shortcut，则解压

不是shortcut直接输出

CH 解压最短路

- CHDataUnit::revert (in 'src/ch_dataunit.cpp' line 246)

```
246 void CHDataUnit::revert(const edge_descriptor_t& ed,
247         std::vector<topo_link_id_t>& id_vec,
248         std::vector<ch::weight_t>& weight_vec) const
249 {
250     const CHGraph& graph = this->graph;
251     if (graph[ed].edge_shortcut_flag == false) {
252          id_vec.push_back(graph[ed].edge_id);
253          weight_vec.push_back(graph[ed].weight);
254
255          /*com_writelog(COMLOG_DEBUG, "    Edge_id: %d, Weight : %d.", */
256                       /*graph[ed].edge_id, graph[ed].weight);*/
257     }
258     else {
259          topo_link_id_t link_id = graph[ed].edge_id;
260          shortcut_edge_t shortcut_edge;
261          this->link_id_to_shortcut(link_id, shortcut_edge);
262
263          edge_descriptor_t first_edge;
264          this->link_id_to_ed(shortcut_edge.first_edge_id, first_edge);
265
266          edge_descriptor_t second_edge;
267          this->link_id_to_ed(shortcut_edge.second_edge_id, second_edge);
268
269          this->revert(first_edge, id_vec, weight_vec);
270          this->revert(second_edge, id_vec, weight_vec);
271     }
272 }
273
```

输入：层次图上的一条边
输出：这条边在原图中的边和边权

这条边不是shortcut，
则直接返回
（思考：这种情况会发生吗？）

这条边是shortcut，
解压变成两条边，
递归解压

真正的 Q&A