Single Source Shortest Path

一、问题定义

单源最短路径（Single Source Shortest Path，SSSP）问题是指在一个加权有向图中，从源节点出发到达所有其他节点的最短距离问题。SSSP_Path问题则在SSSP问题的基础上计算最短路径问题

二、基于GRAPE的分布式算法实现

(一)、语义定义

sssp_path_context.h为sssp问题初始化变量，初始化消息管理器。

(二)、关键函数定义

sssp_path.h定义了GRAPE并行化计算sssp查询的$PEval, IncEval$函数。

```
/*截取的代码删去了原代码中计时的部分*/
void PEval(const fragment_t& frag, context_t& ctx,
             message_manager_t& messages) {
    vertex_t source;
    // 判断source节点是否在本片段，如果在，用source变量指向该节点
    bool native_source = frag.GetInnerVertex(ctx.source_id, source);

    // 局部计算时只考虑查询中source节点到本片段其他节点的最短距离和路径
    if (native_source) {
        ctx.path_distance[source] = 0.0;
        ctx.predecessor[source] = source;
        // 计算source到本片段中其他点的距离并识别F.O中的点，合并消息
        vertexProcess(source, frag, ctx, messages);
    }
    messages.ForceContinue();
}
```

在计算出$source$节点到所在片段的其他点的最短路径的同时，我们也将所有$F.O$的状态发生改变的节点记录在需要传递的消息中，$worker$将该消息传递给$host$，然年后进入增量计算阶段。

```
/*截取的代码删去了原代码中计时的部分*/
void IncEval(const fragment_t& frag, context_t& ctx,
               message_manager_t& messages) {

    auto inner_vertices = frag.InnerVertices();

    vertex_t v, u;
    pair_msg_t msg;
    while (messages.GetMessage<fragment_t, pair_msg_t>(frag, u, msg)) {
        frag.Gid2Vertex(msg.first, v);
        double new_distu = msg.second;
        // 局部计算过程中并没有更新属于F.O的节
        // 点的状态，而是将其保存在了消息中
        if (ctx.path_distance[u] > new_distu) {
            ctx.path_distance[u] = new_distu;
            ctx.predecessor[u] = v;
            ctx.curr_updated.Insert(u);
        }
    }

    ctx.prev_updated.Swap(ctx.curr_updated);
    ctx.curr_updated.Clear();

    // 增量计算的起点是在局部计算中获取的那个F.O中的点
    // 对应到其他片段就是F.I中与source节点所在片段有关联的点
    for (auto v : inner_vertices) {
        if (ctx.prev_updated.Exist(v)) {
            vertexProcess(v, frag, ctx, messages);
        }
    }

    // 如果存在更新，那么向P0传递消息
    if (!ctx.curr_updated.Empty()) {
        messages.ForceContinue();
    }

    //
    vertex_t source;
    bool native_source = frag.GetInnerVertex(ctx.source_id, source);
    size_t row_num = 0;

    for (auto v : inner_vertices) {
        // 如果v不是source节点且source可以到达v
        if (!(native_source && v == source) &&
            ctx.path_distance[v] != std::numeric_limits<double>::max()) {
        row_num++;
        }
    }
    std::vector<oid_t> data;
    std::vector<size_t> shape{row_num, 2};
    //记录路径（实际只记录v和v的前继点）
    for (auto v : inner_vertices) {
        if (!(native_source && v == source) &&
            ctx.path_distance[v] != std::numeric_limits<double>::max()) {
        data.push_back(frag.GetId(ctx.predecessor[v]));
        data.push_back(frag.GetId(v));
        }
    }
    ctx.assign(data, shape);
}
```

增量计算函数将基于局部计算后传递给$host$的消息计算并行$source$到每个片段上的每个节点的最短距离和最短路径，如果增量计算过程中没有更新任何节点的状态分量，那么该片段不向$host$回传消息。

```
// 具体的最短距离计算函数
void vertexProcess(vertex_t v, const fragment_t& frag,
context_t& ctx, message_manager_t& messages) {
    // 获取节点v的出边
    auto oes = frag.GetOutgoingAdjList(v);
    // 获取节点v的编号
    vid_t v_vid = frag.Vertex2Gid(v);
    // 遍历v的出边，
    for (auto& e : oes) {
        auto u = e.get_neighbor();
        double new_distu;
        double edata = 1.0;
        vineyard::static_if<!std::is_same<edata_t, grape::EmptyType>{}>(
            [&](auto& e, auto& data) {
                data = static_cast<double>(e.get_data());
            })(e, edata);
        // 计算到u的距离
        new_distu = ctx.path_distance[v] + edata;
        //如果u属于F.O，那么同步消息；否则更新context中u的状态
        if (frag.IsOuterVertex(u)) {
            messages.
            SyncStateOnOuterVertex<fragment_t, pair_msg_t>(frag,
             u, std::make_pair(v_vid, new_distu));
        } else {
            if (ctx.path_distance[u] > new_distu) {
                // 这里可以看出使用的是Dijkstra算法
                ctx.path_distance[u] = new_distu;
                //保存最短距离时u的前继点，以便后续回溯返回最短路径
                ctx.predecessor[u] = v;
                // 显示传递需要更新的变量
                ctx.curr_updated.Insert(u);
            }
        }
    }
}
```

根据定义好$PEval$和$IncEval$，GRAPE分别初始化各个$worker$和$host$。

```
/*截取的代码删去了原代码中计时的部分*/
template <class... Args>
void Query(Args&&... args) {

    auto& graph = context_->fragment();

    MPI_Barrier(comm_spec_.comm());

    context_->Init(messages_, std::forward<Args>(args)...);

    int round = 0;

    messages_.Start();

    messages_.StartARound();

    // 调用PEval计算source所在片区的单源最短路径
    app_->PEval(graph, *context_, messages_);

    // 同步所有worker的消息
    messages_.FinishARound();

    // 开始执行增量计算，每轮计算同步一次消息，一旦所有worker不在发回消息，计算结束
    while (!messages_.ToTerminate()) {
        t = grape::GetCurrentTime();
        round++;
        messages_.StartARound();

        app_->IncEval(graph, *context_, messages_);

        messages_.FinishARound();
    }

    MPI_Barrier(comm_spec_.comm());

    messages_.Finalize();
}
```