Bug description

The desired functionality is described in #584. `pgr_drivingDistance` returns all nodes that are inside given `drivingDistance`. The problem is that the spanning tree result is missing additional partial edges at leaves that end at inbetween nodes on which the object arrives at exactly the `drivingDistance` value.

Function is commonly used to get the isochrones. The core algorithm is a simple Dijkstra algorithm - not as fast as it could be for the isochrones usecase.

This could be considered a non-bug and the workaround would be to just expand the spanning tree by new edges at the leaves. Depending on how this is done, the expansion could have a lot of unnecessary or overlapping partial edges.

Bugfix description

This section contains a description of how one could support the partial edges in the result.

Function call stack

`many_to_dist_driving_distance.c`

Title file is the entry point of the pgsql driver function `pgr_drivingDistance`. From the arguments it is clear that the function fetches the edges from the database, processes them with the core function `do_pgr_driving_many_to_dist` that directly stores the results in a place that is accessed by the database later. This file does not need any changes for the bugfix.

`drivedist_driver.cpp`

The implementation of `do_pgr_driving_many_to_dist` is in the title file. The fetched edges are used to construct the graph data structure that can be processed by the `boost::graph` library. This might be a possible place of change for the bugfix if different data structure is used for graph or output.

Input and output is documented below:

```c
// inputs
typedef struct {
  int64_t id; // Id of the edge that is marked in comment as E_ID.
  int64_t source; // Id of the starting node.
  int64_t target; // Id of the destination node.
  double cost; // source -> target
  double reverse_cost; // target -> source
                       // (can be -1 -> ignored during graph construction)
} pgr_edge_t;
// outputs
typedef struct {
  int seq; // enumerates the resulting edges of the spanning tree.
  int64_t start_id; // starting vertex (will be repeated multiple times)
  int64_t end_id; // ending node - a leaf in the spanning tree.
  int64_t node;  // this should be the same thing as end_id.
  int64_t edge; // This should be an id E_ID.
  double cost; // cost of going from the predecessor to node using edge.
  double agg_cost; // cost of going from start_id to end_id.
} General_path_element_t;
```

Input can be used to construct all the relevant data structures necessary to run any isochrones finding algorithm. Output can easily be constructed by keeping track of the predecessor in the optimal path (predecessor information can hold a pointer to the edge so that `edge` and `cost` can be copied to the result). `agg_cost` is the distance from `start_id` to `end_id/node`.

The current construction is based on supporting the `boost::graph` interface. Explaining the details of `Basic_vertex`, `Basic_edge` or `graph::Pgr_base_graph` is skipped because it is not relevant for the bugfix.

The function ends up calling `pgr_dijkstra.hpp:pgr_drivingDistance` that is explained in the next subsection.

`pgr_dijkstra.hpp`

Contains various functions that use the `boost::graph::dijkstra_shortest_paths`. The main way of how `drivingDistance` limit is imposed is by the custom implementation of DijkstraVisitor. When a node is visited that has a distance larger than the given `drivingDistance` an exception is thrown and the exploration is stopped. All of the collected edges are then transformed into `basePath_SSEC.cpp:Path` objects that are then transformed into the output `General_path_element_t` later.

Experiments were run during the process of bugfixing and it looked like the construction of `basePath_SSEC.cpp:Path` result might have an unintentional quadratic behavior. Still not confirmed but can easily be profiled in the future.

Given the posibility to provide multiple starting vertices the result of the function is a collection of `basePath_SSEC.cpp:Path` objects. Given that `basePath_SSEC.cpp:Path` contains all the edges necessary for output, the implementation in `drivedist_driver.cpp` transforms the `Path` using `Path:generate_postgres_data` to proper result format and stores it into `return_tuples` and `return_count`.

Where the bug happens

Bug happens in the `pgr_dijkstra.hpp` implementation. The `DijkstraVisitor` is stopping at the first node that is further away than the given `drivingDistance`. This means that the information about the edges that might be partially travelled is lost.

There is no way to adjust the input data or the `DijkstraVisitor` to get the partial edges in the result.

How to fix it

Dirty way

Visiting all the neighbor nodes for each leaf in the spanning tree and marking the `edge` with a lower `agg_cost` to indicate that it is partially traveled and setting `agg_cost = drivingDistance`. This will definitely result in a lot of overlapping edges. This can be considered an incorrect isochrone result. But this allows the fix to work with existing data structure and algorithm.

The Only Way :D

Writing a new dijkstra loop that visits the partial edges properly. Loop still stops when a node further away than the given `drivingDistance` is encountered. There should be no expansion of nodes that are the target on the partially travelled edge.

Implementation is given below. If graphs are dense then an implementation without the queue should be faster (implementation not given). Experiments that were made for dense and tree graphs show that the implementation below is 2-35x faster than `pgr_drivingDistance`. Also, the edge cases discussed in the section below are not supported in the implementation below (but can easily be).

```
// agg_cost at node, node Id
typedef std::tuple<double, int64_t> pq_el;

struct Pred {
  // Can hold edge pointer.
  int64_t edgeId;
  int64_t predId;
  double cost;
};

struct Edge {
  int64_t id;
  int64_t neighborId;
  double cost;
};

std::pair<std::vector<Pred>, std::vector<double>>
dijkstra(size_t n, std::vector<pgr_edge_t> edges, int64_t startVertex,
         double drivingDistance) {
  // Could use Edge* to save on memory.
  std::vector<std::vector<Edge>> adj(n); // adjacency matrix
  for (auto &e : edges) {
    Edge ee;
    ee.id = e.id;
    ee.neighborId = e.target;
    ee.cost = e.cost;
    adj[e.source].push_back(ee);
  }
  std::set<pq_el> q; // priority queue
  std::vector<Pred> predecessors(n);
  std::vector<double> distances(n, std::numeric_limits<double>::infinity());
  predecessors[0] = {0, -1, 0.};
  q.insert({0., startVertex});
  while (!q.empty()) {
    double dist;
    int64_t nodeId;
    std::tie(dist, nodeId) = *q.begin();
    if (dist > drivingDistance) {
      break;
    }
    q.erase(q.begin());
    if (dist == drivingDistance) {
      // If distance is exactly the limit there is no need to go to neighbors.
      continue;
    }
    for (auto &&e : adj[nodeId]) {
      double aggDist = dist + e.cost;
      if (distances[e.neighborId] > aggDist) {
        q.erase({distances[e.neighborId], e.neighborId});
        distances[e.neighborId] =
            aggDist > drivingDistance ? drivingDistance : aggDist;
        predecessors[e.neighborId] = {e.id, nodeId, e.cost};
        q.insert({distances[e.neighborId], e.neighborId});
      }
    }
  }
  return make_pair(predecessors, distances);
}
```

Edge cases

Edge cases are mostly the result of asymmetric properties of the graph. They do not need to be covered if the result is considered correct. Example is given below:

```
Driving distance limit = 15
Time at A = 14 (arrived through another node)
Time at B = 13
A -> B = 8
B -> A = 1

14        8           13
A-----C---------------->B
     15 -- given the limit C is at 1/8 of the A-B edge from A.

14        1           13
A<----C----------------B
     13.875 -- given the B-A edge C is at 7/8 of B-A edge from B.
```

The bugfix given in the document would not return the partial edge describing C. Maybe the edge is not desired because C is reachable from B in less time than `drivingDistance`. Although, if one were to use the graph and traverse to A, then the reachability is obviously at C. On the other hand C on the edge A-B may not even be the same location on the edge B-A so the example above might not be considered an edge case.

There are probably similar examples and they should be resolved with further contemplation.