Causal graph quantum dynamic simulation

auteurs...

1 Introduction

2 Results

3 Tests

We did some tests to validate the simulations, including (but not limitted to) the classical and quantum injectivity tests.

3.1 classical injectivity test

To check if our graph dynamics is reversible, we can apply on a huge number of random graphs a single step, and a reversed step and check that we get back the same graph with started with.

This was tested on arround 10k graphs, for chich the particules position were radomized, but the graph name was kept, making it more complex as each step on random particules positions caused splits and merges.

3.2 quantum injectivity test

To check that our graph dynamic is unitary, we can check that computing an arbitrary amount of steps on an arbitrary initial state (incrissing drastically the number of graphs), and then apply the same number of reversed step, and we check that we get back the state we started with.

Even including floating points errors, we can esaily do this test with around 5 steps, obtain around 7M graphs after the 5 forward iteration, and comme back to the same step within 10^{-6} error in the magnitude.

4 Implementation

The code shown below is a simplification of the actual code [1] (classes are shown without member function or decorators, and code inside loops is represented by functions).

The folllowing classes are used in the implementation:

node names corresponding to the name [2] of a single node.

graph names i.e. ordered set of node.

graph rules which describes the dynamic.

graphs with particules graph name and positions of particules.

superposition of graphs

---

[1]https://github.com/jolatechno/Quantum-graph-simulation.git

[2]arithmetique de nom

4.1 node names

Nodes can either be "left" or "right" containers (pointing to another node), "elemnts" (a simple integer name) or pairs (a merge of two other nodes, pointing to two other nodes).

We mark each node with a "has_most_left_zero_" (which is inerited through merges) variable to keep track of the node that contains the first nodes.

4.1.1 class

```
struct node {
  int left_idx__or_element_;
  int right_idx__or_type_;
  size_t hash_ = 0;
  bool has_most_left_zero_ = false;
  // ...
```

4.1.2 constructors

The following constructor helps construct an "element-type" node (which are just a single integer, and are used to initialize graphs):

```
  // ...
  node(int n) :
    left_idx__or_element_(n), right_idx__or_type_(element_idx) {

    hash_ = n;
    boost::hash_combine(hash_, element_idx);

    has_most_left_zero_ = n == 0;
  }
  // ...
```

We also need a constructor for "left/eright-type" node (noted "u.l" or "u.r"):

```
  // ...
  node(int const idx, node_t const &other, int const type) :
    left_idx__or_element_(idx), right_idx__or_type_(type) {

    boost::hash_combine(hash_, other.hash_);
    boost::hash_combine(hash_, type);

    // check for most left zero
    if (is_left())
      has_most_left_zero_ = other.has_most_left_zero_;
  }
  // ...
```

And finally, we need a constructor to construct "pair-type" nodes (noted "u∧v"):

```
  // ...
  node(int const left_idx, node_t const &left,
    int const right_idx, node_t const &right) :
    left_idx__or_element_(left_idx),
    right_idx__or_type_(right_idx){

    boost::hash_combine(hash_, left.hash_);
    boost::hash_combine(hash_, right.hash_);
    has_most_left_zero_ = left.has_most_left_zero_ || right.has_most_left_zero_;
  }
  // ...
}
```

4.1.3 hash

the hash of a node is stored in the "hash_" varibale, and is simply calculated by the constructor (since it never changes).

To compute the hash of a node, we simply combine the hash of all children nodes or of the integer representing the element, and/or the type of the node.

4.2 graph names

To store the list of nodes, we use a main nodes list, which we keep in a lexicographical order (the first node should always be the node with the "most left zero").

Other nodes which are pointed to are stored into a buffer vector, for which we use grabage collection.

4.2.1 class

```cpp
class graph_name {
private:
  // node list
  std::vector<node_t> nodes_;
  std::vector<node_t> node_buff_;
  std::vector<int> trash_collection_;

  // hash
  size_t mutable hash_ = 0;
  bool mutable hashed_ = false;

  //...
```

4.2.2 usefull functions

Since we use trash collection to manage memory, we need to implement a collection to insert a node into the "buffer vector":

```cpp
  //...
  int push_to_buffer(node_t &n) {
    if (trash_collection_.empty()) {
      node_buff_.push_back(n);
      return node_buff_.size() - 1;
    }

    int buff_idx = trash_collection_.back();
    trash_collection_.pop_back();
    node_buff_[buff_idx] = n;
    return buff_idx;
  }
  //...
```

4.2.3 split

```cpp
  //...
  bool inline split(unsigned int idx) {
    hashed_ = false;

    node_t node = nodes_[idx];
    if (node.is_pair()) {
      // read indexes
      int const left_idx = node.left_idx();
      int const right_idx = node.right_idx();
```

```
10
11        // add left node
12        nodes_[idx] = node_buff_[left_idx];
13        trash_collection_.push_back(left_idx);
14
15        // add right node
16        nodes_.insert(nodes_.begin() + idx + 1, node_buff_[right_idx]);
17        trash_collection_.push_back(right_idx);
18
19        // check if we should rotate
20        if (idx == 0 && nodes_[1].has_most_left_zero()) {
21          std::rotate(nodes_.begin(), nodes_.begin() + 1, nodes_.end());
22          return true;
23        }
24
25        return false;
26      }
27
28      // add current node to buffer
29      int buff_idx = push_to_buffer(node);
30
31      // add left node
32      nodes_[idx] = node_t(buff_idx, node, point_l_idx);
33
34      // add right node
35      nodes_.emplace(nodes_.begin() + idx + 1, buff_idx, node, point_r_idx);
36
37      return false;
38    }
39  //...
```

The boolean return argument of this function is used to check if the merge requires a rotation of nodes.

4.2.4   merges

```
1   //...
2   void inline merge(unsigned int idx) {
3     hashed_ = false;
4
5     // destination idx
6     unsigned int right_idx = (idx + 1) % size();
7     auto left = nodes_[idx];
8     auto right = nodes_[right_idx];
9
10     if (left.is_left() && right.is_right()) {
11       int const left_left = left.left_idx();
12       int const right_left = right.left_idx();
13
14       if (node_buff_[left_left].hash() == node_buff_[right_left].hash()) {
15         // trash collect
16         trash_collection_.push_back(left_left);
17         if (left_left != right_left)
18           trash_collection_.push_back(right_left);
19
20         //add node
21         nodes_[right_idx] = node_buff_[left_left];
22         nodes_.erase(nodes_.begin() + idx);
23         return;
24       }
25     }
26
27     //add node
28     int const left_idx = push_to_buffer(left);
29     int const right_idx_ = push_to_buffer(right);
30
```

```
31      nodes_[right_idx] = node_t(left_idx, left, right_idx_, right);
32      nodes_.erase(nodes_.begin() + idx);
33    }
34    //...
```

4.2.5   hash()

To hash a graph name, we simply combine the hash of all the nodes.

```
1    //...
2    size_t inline hash() const {
3      if (hashed_)
4        return hash_;
5
6      hash_ = 0;
7
8      /* combine all nodes hash */
9      for (auto &n : nodes_)
10       boost::hash_combine(hash_, n.hash());
11
12     hashed_ = true;
13     return hash_;
14   }
15   //...
16 }
```

4.3   graphs with particules

A graph is represented by a graph name, and an ordered list of the node's index of particules going both ways.

4.3.1   class

```
1  class graph {
2    // split_merge type enum
3      typedef enum op_type {
4      split_t,
5      merge_t,
6      erase_t,
7      create_t,
8    } op_type_t;
9
10   // typedef of the pair of an index and a op_type
11     typedef std::pair<unsigned int, op_type_t> op_t;
12
13   // variables
14   graph_name_t* name_;
15   std::vector<unsigned int> left_;
16   std::vector<unsigned int> right_;
17
18   // hash
19   size_t mutable hash_ = 0;
20   bool mutable hashed_ = false;
21
22   //...
```

4.3.2   step and reversed_step()

The fonction to step all particules uses the fonctions "rotate_once_left" and "rotate_once_right" which rotate particules if they overflow:

```
1    //...
2    void inline step() {
```

```
3        hashed_ = false;
4
5        rotate_once_right(right_);
6        rotate_once_left(left_);
7     }
8     //...
```

To implement the reverse stpe we simply exchange the roles of "left" and "right" in the "step" function:

```
1   //...
2     void inline reversed_step() {
3        hashed_ = false;
4
5        rotate_once_right(left_);
6        rotate_once_left(right_);
7     }
8     //...
```

4.3.3   split_merge()

```
1   //...
2   void split_merge(std::vector<split_merge_t>& split_merge) {
3     if (split_merge.empty())
4        return;
5
6     hashed_ = false;
7
8     // check for last merge
9     if (split_merge.back() == split_merge_t(size() - 1, merge_t)) {
10       name_->merge(size() - 1);
11       overflow_right(left_);
12       split_merge.pop_back();
13    }
14
15    // check for first element split
16    bool first_split = false;
17
18    // and calculating max displacement
19    int total_displacement = 0;
20    for (auto & [pos, type] : split_merge | std::ranges::views::reverse)
21      if (type == split_t) {
22         ++total_displacement;
23      } else
24         --total_displacement;
25
26    // reserve space for nodes
27    name_->reserve(total_displacement);
28
29    // split and merge names
30    for (auto & [pos, type] : split_merge | std::ranges::views::reverse)
31      if (type == split_t) {
32         first_split |= name_->split(pos);
33      } else
34         name_->merge(pos);
35
36    // move particules
37    auto const split_merge_begin = split_merge.rend();
38    int displacement = total_displacement;
39    auto split_merge_it = split_merge.rbegin();
40    for (auto &left_it : left_ | std::ranges::views::reverse) {
41      // check if there are any nodes left
42      for (;split_merge_it->first >= left_it && split_merge_it != split_merge_begin; ++
       split_merge_it)
43         // check if the node is split or merged
44         if (split_merge_it->second == split_t) {
```

```
45            −−displacement; // decrement the displacement
46          } else
47            ++displacement; // decrement the displacement
48
49      //check if we can exit
50      if (split_merge_it == split_merge_begin && displacement == 0)
51        break;
52
53      left_it += displacement;
54    }
55
56
57    /* we than do the exact same thing with particule going to the right, rplacing ">=" at line
   42 by a ">" */
58
59    // finish first split
60    if (first_split) {
61      rotate_once_left(left_);
62      rotate_once_left(right_);
63    }
64  }
65  //...
```

4.3.4   erase_create()

```
1
2  void inline graph::erase_create(std::vector<op_t>& erase_create) {
3    //...
4    void inline erase_create(std::vector<op_t>& erase_create) {
5      auto const erase_create_begin = erase_create.rend();
6
7      // add and remove left particules
8      auto erase_create_it = erase_create.rbegin();
9      int left_idx = left.size() − 1;
10     if (!left.empty())
11       for (; erase_create_it < erase_create_begin; ++erase_create_it)
12         if (erase_create_it−>second == create_t) {
13           // find the position to insert
14           for (; left[left_idx] >= erase_create_it−>first && left_idx >= 0; −−left_idx) {}
15
16           if (left_idx < 0)
17             break;
18
19           // insert
20           left.insert(left.begin() + left_idx + 1, erase_create_it−>first);
21
22         } else {
23           // find the particule to erase
24           for (; left[left_idx] > erase_create_it−>first && left_idx >= 0; −−left_idx) {}
25
26           if (left_idx < 0)
27             break;
28
29           // erase
30           left.erase(left.begin() + left_idx);
31           −−left_idx;
32         }
33
34       // add the particule at the begin
35       for (; erase_create_it < erase_create_begin; ++erase_create_it)
36         left.insert(left.begin(), erase_create_it−>first);
37
38       /* add and remove right particules */
39   }
```

```
40    // ...
```

4.3.5   hash()

To hash graphs we simply combine all the positions of partricules going in one directions, a separator, the paricules going the other, a separator way and finally the hash of the graph's name.

```
1     // ...
2     size_t hash() const {
3       if (hashed_)
4         return hash_;
5
6       hash_ = 0;
7       boost::hash<unsigned int> hasher;
8
9       // left hash
10      for (auto &l : left_)
11        boost::hash_combine(hash_, l);
12
13       // separator
14        boost::hash_combine(hash_, separator);
15
16        // right hash
17        for (auto &r : right_)
18          boost::hash_combine(hash_, r);
19
20        // separator
21        boost::hash_combine(hash_, separator);
22
23        // name hash
24        boost::hash_combine(hash_, name_->hash());
25
26
27      hashed_ = true;
28      return hash_;
29    }
30    // ...
31 }
```

4.4   graph rules

4.4.1   get_split_merges()

To get the list of all indeces of splits and merges, we iterate through both the "left" and "right" positions, iterating the one that is the smallest so that it always converges towars positions that match (or are one apart for merges):

```
1  /* function to get all split and merges */
2  std::vector<graph::split_merge_t> inline get_split_merge(graph_t* graph) {
3    std::vector<graph::op_t> split_merge;
4
5    unsigned int const last_idx = graph->size() − 1;
6
7    auto left_it = graph->left().begin();
8    auto right_it = graph->right().begin();
9    bool first_or_last_split = *left_it == *right_it && *left_it == 0;
10   first_or_last_split |= graph->left().back() == last_idx && graph->right().back() == last_idx;
11
12   while (right_it < right_end = graph->right().end() && left_it < graph->left().end())
13     if (*left_it == *right_it) {
14       // check for split
15       split_merge.push_back({*left_it, graph->split_t});
16
17       ++right_it;
18       ++left_it;
```

8

```
19      } else if (*left_it < *right_it) {
20        // chedk for merges
21        if (*left_it == *right_it - 1)
22          if (left_it == graph->left().end() - 1) {
23            split_merge.push_back({*left_it, graph->merge_t});
24          } else if (*(left_it + 1) != *right_it)
25            split_merge.push_back({*left_it, graph->merge_t});
26
27        ++left_it;
28      } else
29        ++right_it;
30
31    if (!first_or_last_split)
32      if (graph->left().back() == last_idx && graph->right()[0] == 0)
33        split_merge.push_back({last_idx, graph->merge_t});
34
35    return split_merge;
36 }
```

4.4.2 get_erase_create()

```
1 std::vector<graph::op_t> inline get_erase_create(graph_t const* graph) {
2    std::vector<graph::op_t> create_erase;
3
4    unsigned int const last_idx = graph->size() - 1;
5
6    auto left_it = graph->left.begin();
7    auto right_it = graph->right.begin();
8
9    unsigned int last_pos = 0;
10
11   auto const left_end = graph->left.end();
12   auto const right_end = graph->right.end();
13   while (right_it < right_end && left_it < left_end) {
14     // add create
15     unsigned int next_last_pos = std::min(*left_it, *right_it);
16     for (unsigned int i = last_pos; i < next_last_pos; ++i)
17       create_erase.push_back({i, graph->create_t});
18
19     // update lest_pos
20     last_pos = next_last_pos + 1;
21
22     if (*left_it == *right_it) {
23       // check for erase
24       create_erase.push_back({*left_it, graph->erase_t});
25
26       ++right_it;
27       ++left_it;
28     } else {
29       if (*left_it < *right_it) {
30         ++left_it;
31       } else
32         ++right_it;
33     }
34   }
35
36   // complete the unfinished list
37   auto it = left_it == left_end ? right_it : left_it;
38   auto const end = left_it == left_end ? right_end : left_end;
39   for (; it < end; ++it) {
40     // add create
41     unsigned int next_last_pos = *it;
42     for (unsigned int i = last_pos; i < next_last_pos; ++i)
43       create_erase.push_back({i, graph->create_t});
```

```
44
45      // update lest_pos
46      last_pos = next_last_pos + 1;
47   }
48
49   // add create at the end
50   unsigned int const size = graph->size();
51   for (unsigned int i = last_pos; i < size; ++i)
52      create_erase.push_back({i, graph->create_t});
53
54   return create_erase;
55 }
```

4.5   superposition of graphs

4.5.1   class

To allow parallel insertions of graphs into the lists of all graphs, we use tbb concurent hashmap.

We make the approximation that graphs with the same hash are equals and should therefor interact.

```
1  class state {
2    // hasher
3    struct graph_hasher { /* */ }
4
5    // comparator
6    struct graph_comparator { /* */ }
7
8    // type definition
9    typedef tbb::concurrent_unordered_multimap<graph_t*, std::complex<long double>, graph_hasher,
        graph_comparator> graph_map_t;
10
11   // main list
12   graph_map_t graphs_;
13
14   // parameters
15   std::complex<long double> non_merge_ = -1;
16   std::complex<long double> merge_ = 0;
17
18   //..
```

4.5.2   reduce_all()

```
1    //...
2    void reduce_all() {
3      for(auto it = graphs_.begin(); it != graphs_.end();) {
4        // range of similar graphs to delete
5        auto upper = graphs_.equal_range(it->first).second;
6
7        for(auto jt = std::next(it); jt != upper; ++jt)
8            it->second += jt->second;
9
10       // if the first graphgs has a zero probability, erase the whole range
11       if (!check_zero(it->second))
12         ++it;
13
14       it = graphs_.unsafe_erase(it, upper);
15     }
16   }
17   //...
```

4.5.3 step_all()

We pass a function "rule" that returns a vector of graphs and magnitude obtained from a single graph.

```cpp
// ...
void state::step_all(std::function<tbb::concurrent_vector<std::pair<graph_t*, std::complex<long
    double>>>(graph_t* g)> rule) {
  graph_map_t buff;
  buff.swap(graphs_);

  #pragma omp parallel
  #pragma omp single
  for (auto & [graph, mag] : buff)
  #pragma omp task
  {
    auto graphs = rule(graph);
    for (auto & [graph_, mag_] : graphs)
    #pragma task
        graphs_.insert({graph_, mag_ * mag});
  }
}
// ...
}
```

4.5.4 source of errors