# 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 drasticly 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 200k graphs after the 5 forward iteration, and comme back to the same step within 0.001% 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.

**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();
```

```

    // add left node
    nodes_[idx] = node_buff_[left_idx];
    trash_collection_.push_back(left_idx);

    // add right node
    nodes_.insert(nodes_.begin() + idx + 1, node_buff_[right_idx]);
    trash_collection_.push_back(right_idx);

    // check if we should rotate
    if (idx == 0 && nodes_[1].has_most_left_zero()) {
      std::rotate(nodes_.begin(), nodes_.begin() + 1, nodes_.end());
      return true;
    }

    return false;
  }

  // add current node to buffer
  int buff_idx = push_to_buffer(node);

  // add left node
  nodes_[idx] = node_t(buff_idx, node, point_l_idx);

  // add right node
  nodes_.insert(nodes_.begin() + idx + 1,
    node_t(buff_idx, node, point_r_idx));

  return false;
}
//...
```

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

### 4.2.4 merges

```
//...
void inline merge(unsigned int idx) {
  hashed_ = false;

  // destination idx
  unsigned int right_idx = (idx + 1) % size();
  auto left = nodes_[idx];
  auto right = nodes_[right_idx];

  if (left.is_left() && right.is_right()) {
    int const left_left = left.left_idx();
    int const right_left = right.left_idx();

    if (node_buff_[left_left].hash() == node_buff_[right_left].hash()) {
      // trash collect
      trash_collection_.push_back(left_left);
      if (left_left != right_left)
        trash_collection_.push_back(right_left);

      //add node
      nodes_[right_idx] = node_buff_[left_left];
      nodes_.erase(nodes_.begin() + idx);
      return;
    }
  }

  //add node
  int const left_idx = push_to_buffer(left);
  int const right_idx_ = push_to_buffer(right);
```

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

### 4.2.5 hash()

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

```
  //...
  size_t inline hash() const {
    if (hashed_)
      return hash_;

    hash_ = 0;

    /* combine all nodes hash */
    for (auto &n : nodes_)
      boost::hash_combine(hash_, n.hash());

    hashed_ = true;
    return hash_;
  }
  //...
}
```

## 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

```
class graph {
  // variables
  graph_name_t* name_;
  std::vector<unsigned int> left_;
  std::vector<unsigned int> right_;

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

  //...
```

### 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:

```
  //...
  void inline step() {
      hashed_ = false;

      rotate_once_right(right_);
      rotate_once_left(left_);
    }
    //...
```

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

```
//...
  void inline reversed_step() {
    hashed_ = false;

    rotate_once_right(left_);
    rotate_once_left(right_);
  }
  //...
```

### 4.3.3 split_merge()

```
//...
void split_merge(std::vector<split_merge_t>& split_merge) {
  /* first check if there are any split or merges */

  hashed_ = false;

  // check for last merge
  /* ... */
  if (last_merge()) {
    name_->merge(last_idx);
    overflow_right(left_);
    split_merge.pop_back();
    /* ... */
  }

  // check for first element split
  bool first_split = false;

  // split and merge names
  // and calculating max displacement
  int total_displacement = 0;
  for (auto split_merge_it = split_merge.rbegin(); split_merge_it != split_merge.rend(); ++split_merge_it)
    if ((*split_merge_it).second == split_t) {
      first_split |= name_->split((*split_merge_it).first) && (*split_merge_it).first == 0;

      ++total_displacement;
    } else {
      name_->merge((*split_merge_it).first);
      --total_displacement;
    }

  // move particules
  int displacement = total_displacement;
  auto split_merge_it = last_split_merge;
  for (auto left_it = left_.rbegin(); left_it != left_.rend(); ++left_it) {
    // check if there are any nodes left
    while (split_merge_it >= split_merge.begin()) {
      // check if the node is split or merged
      if ((*split_merge_it).first >= *left_it) {
        if ((*split_merge_it).second == split_t) {
          --displacement; // decrement the displacement
        } else
          ++displacement; // decrement the displacement
        --split_merge_it;
      } else
        break;

    }

    if (split_merge_it == split_merge.begin() && displacement == 0)
      break;

```

```
      *left_it += displacement;
    }

    /* we than do the exact same thing with particule going to the right, rplacing ">=" at line
    48 by a ">" */

    // finish first split
    if (first_split) {
      rotate_once_left(left_);
      rotate_once_left(right_);
    }
  }
  //...
```

### 4.3.4 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.

```
  //...
  size_t hash() const {
    if (hashed_)
      return hash_;

    hash_ = 0;
    boost::hash<unsigned int> hasher;

    // left hash
    for (auto &l : left_)
      boost::hash_combine(hash_, l);

     // separator
      boost::hash_combine(hash_, separator);

      // right hash
      for (auto &r : right_)
        boost::hash_combine(hash_, r);

      // separator
      boost::hash_combine(hash_, separator);

      // name hash
      boost::hash_combine(hash_, name_->hash());


    hashed_ = true;
    return hash_;
  }
  //...
}
```

### 4.3.5 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):

```
/* function to get all split and merges */
std::vector<graph::split_merge_t> inline get_split_merge(graph_t* graph) {
  std::vector<graph::split_merge_t> split_merge;

  /* first check if there are any particules to check */

  /* definition of left_it, right_it, last_left, last_right ... */

```

```
  bool first_or_last_split = *left_it == *right_it && *left_it == 0;
  while (true) {
    /* check if there are any particules left after iterating */
    if (right_it > last_right ||
    left_it > last_left)
      break;

    if (*left_it == *right_it) {
      /* check for split */
      split_merge.push_back({(*left_it), graph->split_t});

      /* check for last split */
      if (*right_it == last_idx)
        first_or_last_split = true;

      ++right_it;
      ++left_it;
    } else if (*left_it < *right_it) {
      /* chedk for merges */
      if (*left_it == *right_it - 1)
        if (left_it == last_left) {
          split_merge.push_back({(*left_it), graph->merge_t});
        } else if (*(left_it + 1) != *right_it)
          split_merge.push_back({(*left_it), graph->merge_t});

      ++left_it;
    } else
      ++right_it;
  }

  if (!first_or_last_split)
    if (*last_left == last_idx &&
    graph->right()[0] == 0)
      split_merge.push_back({last_idx, graph->merge_t});

  return split_merge;
}
```

## 4.4 superposition of graphs

### 4.4.1 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.

```
class state {
  // hasher
  struct graph_hasher { /* */ }

  // comparator
  struct graph_comparator { /* */ }

  // type definition
  typedef tbb::concurrent_unordered_multimap<graph_t*, std::complex<long double>, graph_hasher,
    graph_comparator> graph_map_t;

  // main list
  graph_map_t graphs_;

  // parameters
  std::complex<long double> non_merge_ = -1;
  std::complex<long double> merge_ = 0;

  //..
```

### 4.4.2 subset of splist and merges

We first need to know the number of possible graphs from the lists of splits and merges:

```
template<class T>
int num_subset(std::vector<T> const &vect) {
  return pow(2, vect.size());
}
```

We can then associate a number between 0 and "num_subset" to a certain compositions of all the splits and merges (by seeing the nth binary ditgit of this number as enabling the nth split or merge) :

```
std::pair<std::vector<graph::split_merge_t>, std::complex<long double>> subset(std::vector<graph::split_merge_t>& split_merge, int subset_numb,
  std::complex<long double>& non_merge, std::complex<long double>& merge) {

  std::vector<graph::split_merge_t> res;
  std::complex<long double> proba = 1;

  for (int i = 0; i < split_merge.size(); ++i) {
    long double sign = 1 - 2*split_merge[i].second;

    /* i_th bit of subset_numb */
    if (subset_numb%2) {
      /* add opperation to res */
      res.push_back(split_merge[i]);

      /* get proba */
      proba *= std::complex<long double>(merge.real(), sign*merge.imag());
    } else
      proba *= sign*non_merge;


    subset_numb /= 2;
  }

  return {res, proba};
}
```

### 4.4.3 reduce_all()

```
  //...
  void reduce_all() {
    for(auto it = graphs_.begin(); it != graphs_.end();) {
      // range of similar graphs to delete
      auto range = graphs_.equal_range(it->first);
      auto start = range.first;

      for(auto jt = std::next(it); jt != range.second; ++jt)
          it->second += jt->second;

      // if the first graphgs has a zero probability, erase the whole range
      if (!check_zero(it->second))
        ++start;

      it = graphs_.unsafe_erase(start, range.second);
  }
  //...
```

### 4.4.4 step_split_merge_all() and reversed_split_merge_step_all()

```
  //...
  void step_split_merge_all() {
    graph_map_t buff;
```

```
    buff.swap(graphs_);

    #pragma omp parallel
    #pragma omp single
    for (auto & [graph, mag] : buff)
    #pragma omp task
      {

        graph->step(); //graph->reversed_step(); for reversed_split_merge_step_all()

        auto split_merge = get_split_merge(graph);

        #pragma omp task
        {
          // update the probability of the graph without split or merge
        auto [_, mag_no_split_merge] = subset(split_merge, 0, non_merge_, merge_);
        graphs_.insert({graph, mag * mag_no_split_merge});
        }

        // add all graphs that actually have some split ot merge
        const int n_max = num_subset(split_merge);
        #pragma omp taskloop
        for (int j = 1; j < n_max; ++j) {
          graph_t* g_ = graph->copy();

          auto [split_merge_list, mag_split_merge] = subset(split_merge, j, non_merge_, merge_);
          g_->split_merge(split_merge_list);

          //add graph
          graphs_.insert({g_, mag * mag_split_merge});
        }
      }
  }
  //...
```

### 4.4.5 source of errors