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 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.

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

```
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":

```
  //...
  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

```
  //...
  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_.emplace(nodes_.begin() + idx + 1, 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 {
  // split_merge type enum
    typedef enum op_type {
    split_t,
    merge_t,
    erase_t,
    create_t,
  } op_type_t;

  // typedef of the pair of an index and a op_type
    typedef std::pair<unsigned int, op_type_t> op_t;

  // variables
  graph_name_t name_;
  std::vector<char /*bool*/> left;
  std::vector<char /*bool*/> right;

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

  //...
```

4.3.2 step and reversed_step()

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

      std::rotate(left.begin(), left.begin() + 1, left.end());
```

```
    std::rotate(right.rbegin(), right.rbegin() + 1, right.rend());
    }
    //...
```

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

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

      std::rotate(left.rbegin(), left.rbegin() + 1, left.rend());
    std::rotate(right.begin(), right.begin() + 1, right.end());
    }
    //...
```

4.3.3 split_merge()

```
  //...
  void split_merge(std::vector<split_merge_t>& split_merge) {
    if (split_merge.empty())
      return;

    hashed_ = false;

    bool first_split = false;
    auto const size_ = size();
    for (auto & [pos, type] : split_merge | std::ranges::views::reverse)
      if (type == split_t) {
        first_split |= name_.split(pos);

        // change right
        right[pos] = false;

        //insert
        right.insert(right.begin() + pos + 1, true);
        left.insert(left.begin() + pos + 1, false);
      } else if (type == merge_t) {
        name_.merge(pos);

        // next pos
        unsigned short int next_pos = (pos + 1) % size_;

        // change left
        left[next_pos] = true;

        //erase
        right.erase(right.begin() + pos);
        left.erase(left.begin() + pos);
      }

    /* finish first split */
    if (first_split) {
      std::rotate(left.begin(), left.begin() + 1, left.end());
      std::rotate(right.begin(), right.begin() + 1, right.end());
    }
  }
  //...
```

4.3.4 erase_create()

```

void inline graph::erase_create(std::vector<op_t>& erase_create) {
  //...
  void inline erase_create(std::vector<op_t>& erase_create) {
```

```
    if (erase_create.empty())
      return;

    hashed_ = false;

    for (auto & [pos, type] : erase_create)
      if (type == erase_t) {
        left[pos] = false;
        right[pos] = false;
      } else if (type == create_t) {
        left[pos] = true;
        right[pos] = true;
      }
  }
  //...
```

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.

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

    // save memory
    left.shrink_to_fit();
    right.shrink_to_fit();

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

    // left hash
    auto const size_ = size();
    for (short int i = 0; i < size_; ++i)
      if (left[i])
        boost::hash_combine(hash_, i);

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

      // right hash
      for (short int i = 0; i < size_; ++i)
      if (right[i])
        boost::hash_combine(hash_, i);

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

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

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

4.3.6 get_split_merges()

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

```
  std::vector<graph::op_t> split_merges;

  auto const size_ = graph.size();
  for (int i = 0; i < size_; ++i)
    if (graph.left[i] && graph.right[i]) {
      split_merges.push_back({i, graph.split_t});
    } else {
      // next pos
      unsigned short int next_pos = (i + 1) % size_;

      if (graph.left[i] && graph.right[next_pos] && !graph.left[next_pos])
        split_merges.push_back({i, graph.merge_t});
    }

  return split_merges;
}
```

4.3.7 get_erase_create()

```
std::vector<graph::op_t> inline get_erase_create(graph_t const &graph) {
  std::vector<graph::op_t> create_erases;

  for (int i = 0; i < graph.size(); ++i)
    if (graph.left[i] && graph.right[i]) {
      create_erases.push_back({i, graph.erase_t});
    } else if (!graph.left[i] && !graph.right[i])
      create_erases.push_back({i, graph.create_t});

  return create_erases;
}
```

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 reduce_all()

```cpp
//...
void reduce_all() {
  graph_map_t buff; // faster, parallel reduce that uses WAY more ram
  buff.swap(graphs_);

  #pragma omp parallel
  #pragma omp single
    for(auto it = buff.begin(); it != buff.end();) {
      // range of similar graphs to delete
      auto const graph = it->first;
      auto const range = buff.equal_range(graph);

      // next iteration
      it = range.second;

      #pragma omp task
      {
        std::complex<long double> acc = 0;
        for(auto jt = range.first; jt != range.second; ++jt)
            acc += jt->second;

        // if the first graphgs has a zero probability, erase the whole range
        if (!check_zero(acc))
          graphs_.insert({graph, acc});
      }
    }
}
//...
```

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

4.4.4 source of errors