David Spitz
May 28, 2023

ZX Graph Adjacency List to Block Structure Translation

- **What I made**
  - https://github.com/davidbspitz/VyZX_GraphTranslation/blob/main/src/Graph/GraphTranslation.v
    - All code is in this file, which is on a fork from VyZX
  - Translation function from adjacency list representation of a ZX diagram into a semantically equivalent block structure representation
    - Large sub-problem: constructing swap structures between arbitrary permutations from only adjacent swaps, which amounts to an implementation of bubblesort
- **How the algorithms work**
  - *Input type*
    - The adjacency list representation is a record *zx_graph* with three disjoint lists: inputs, outputs, and nodes, which comprise the order of input/output "nodes" (ZX diagrams are open ended so these aren't nodes in the usual sense), and the regular spider nodes. All elements of these lists are assumed to be assigned unique *nat* ids. Then there is an adjacency list of pairs of *nat* which signifies an edge between the two nodes. The record also includes a mapping list which can map *nat* values to the actual spider information for nodes.
    - The input is assumed to be well-formed.
      - Edges must be between existing nodes in the type
      - All normal non-boundary nodes are mapped
      - All *nat* ids are unique, and inputs, outputs, and nodes are disjoint sets
      - I/O nodes must connect to 1 node, and cannot have a self-loop
        - Note that regular nodes can connect themselves to more than one I/O, but each I/O can only connect to one other node
      - The order of I/O lists is correct from the original ZX graphical diagram
    - The input is assumed to not contain Hadamard edges.
  - *Output type*
    - ZX (length inputs) (length outputs)
  - *Conversion*
    - The goal is to preserve connectivity information as well as input and output order from the input graph, which ensures semantic equivalence

        - This is accomplished by fenceposting in which the current edge information of a ZX diagram is annotated with an ordered list that holds connection information.
            - One way to picture this is to impose the circuit structure onto a ZX diagram, such that if you have *m* outgoing edges, we can imagine this as a stack of *m* wires on top of each other. Then annotate each wire with the node that it is connected to, which could be an input node or a regular node. This makes up a fencepost.
        - The first step of the algorithm is processing the order of input nodes as the first fencepost.
        - Then, we choose the next node to process. We look at all of its neighboring nodes and look at which of those nodes are contained in the current fencepost. Those nodes will serve as inputs to this spider, and the other neighbors will be passed as outputs. In general for regular nodes, whether or not their connections are passed into them or out of them is irrelevant as only connectivity matters (of course not in the case of connections to I/O nodes), so as long as they are eventually connected it is ok.
        - With this information, we construct a goal fencepost which moves the nodes to be connected in the current fencepost to the top, and leaves the rest to the bottom. This is done so the connections can be contiguous so if we put a spider next to them it will properly connect
        - We construct a swap between the current fencepost and the goal, two permutations, which we hand to the bubblesort algorithm to generate
            - This subroutine also assumes that the lists are permutations of each other to work properly
        - Then we connect this swap structure to a node structure which holds the corresponding spider information. This node structure connects the node to the proper input edges along with wires to let the rest of the current connections pass on
        - Lastly we calculate the next fencepost, which removes the connected elements of the fencepost and places *n* copies of the node id, where *n* is the number of outputs of the node
        - Now we have completed the connection process for one node. We continue repeating these steps until all nodes are connected and then we pass one final swap to the ordered output fencepost
    - *Why this should work*
        - In order for this algorithm to be correct, it only needs to preserve I/O order as well as all connections. Extraneous swaps are irrelevant.
            - I/O order is preserved assuming the lists are given in correct order

            - Connectivity is preserved
                - As a base case, consider a diagram with no regular nodes. The algorithm just swaps between the input and output states and connects them properly
                - For any node, if its neighbors appeared already in the algorithm then their edges to it will be in the current fencepost, and they will be connected with the swap and node structure. All nodes will be eventually connected correctly, as for each following node until the outputs are reached all necessary connections will be made, until all that is remaining are those connections left to outputs which are swapped into the correct place.
    - *Annoying cases*
        - Self-loops between regular nodes
            - These are accounted for by a subroutine called for each node structure
        - Inputs connected to inputs, or outputs connected to outputs
            - Pre/post processing phase
            - This is legal, consider just a cup or cap for instance.
            - This is accounted for by a similar swap structure and then replacing all of these with caps or cups
            - By assumption the only way this could occur would be two inputs/outputs connected, as they only have one connection allowed
    - *Evaluation*
        - In many cases here casting is necessary so the term will not fully simplify without other tactics such as *simpl_casts*. Computing is not too slow, but using the tactics takes a long time even with small examples
        - This means that testing has been difficult to do and there could be bugs remaining
- **What is remaining to be done**
    - Proof of the bubblesort swap structure's semantic correctness using QuantumLib's definition of permutations and permutation matrices
    - Translation from block structure to adj list, and proof of bijection
    - Better evaluation mechanism for the translation algorithm, either by optimizing the code or writing cleaner tactics
        - The current unfolding and rewriting is quite slow
        - Then properly testing the translation with this evaluation mechanism
    - Improving the translation algorithm
        - Use of dummy outputs in the translation

            - Since the input type is not well-formed by default, sometimes dummy things are used to make casting go through. This may not be best idea and could be replaced with an overall *option* type instead so errors are clear
        - Good practices for casting and lemmas
            - I'd defer to Adrian and Ben on this as I'm not sure what makes cleaner simplification
        - Could add Hadamard edge support
- **Code sources**
    - I got the bubble sort algorithm on lists from here:
        - https://codeberg.org/mathprocessing/coq-examples/src/branch/master/sorts/bubblesort.v
        - I think originates from SF, under MIT
        - Another version here with verification is possible under MIT
            - https://github.com/holmuk/Sorticoq/blob/master/src/BubbleSort.v