Cirq merge operations algorithm

I spent some time trying to understand merge_k_qubit_unitaries in order to improve it.
The core of the algorithm is implemented in transformer_primitives.merge_operations.

How does merge_operations work?

I asked Copilot to explain the implementation but it didn't give a good answer 🙂

Merge_operations traverses the circuit moment by moment and merges two operations op1 and op2 subject to the following conditions:

1. merge_func(op1, op2) doesn't return None (and in fact returns the merged operation)
2. there is no other operation between op1 and op2 in the circuit
3. Either op1.qubits ⊆ op2.qubits or op2.qubits ⊆ op1.qubits

Let's ignore condition 1) for the moment: assume merge_func always succeeds.

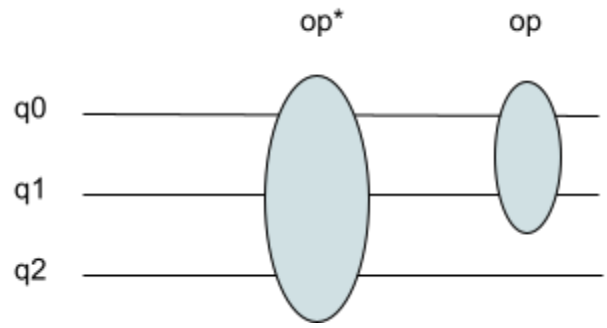
Let op be the current operation we are trying to merge. The algorithm finds the previous moment M with mergeable operations. From 2) and 3), this means:

- M has an operation op* and op* and op share at least one qubit;
- between M and op moment, there is no other moment that has an operation acting on one of op qubits, or one of op measurement or control keys.

Let L = {op*: op* is in M and op* shares at least one qubit with op}.

Next the implementation considers two cases.

Case 1: There is only one element L = {op*} and op.qubits ⊆ op*.qubits
Then op is merged into op*: the new CircuitOperation is in moment M, but now incorporates both op* and op (in this order).



Case 2: Case 1 condition is not true but L is non-empty

Then the algorithm tries to repeatedly compute L, and then pick an operation op* in L and merge it with op, putting the result in op moment.
This requires:

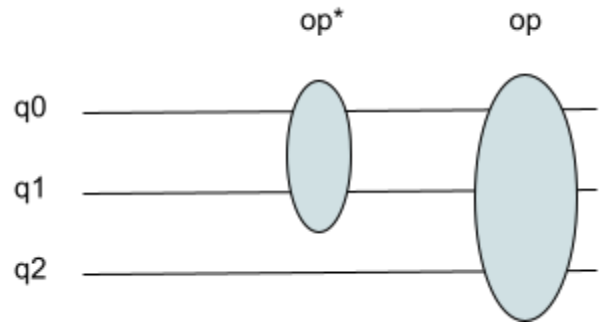
- op*.qubits ⊆ op.qubits

However, the algorithm can hit a snag: if op* can't be merged with op, for example:

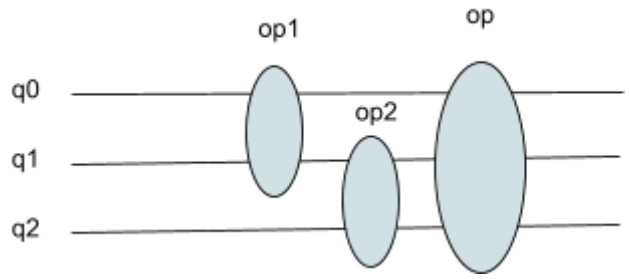
- op*.qubits - op.qubits ≠ ∅

then the qubits used by both op and op* can't be considered anymore: op* will always be in a moment between op and any other operation that uses the same qubit.
If L is exhausted, then another moment is picked, L is recomputed, until there is no qubit left from op.qubits that can potentially harbor a mergeable operation.

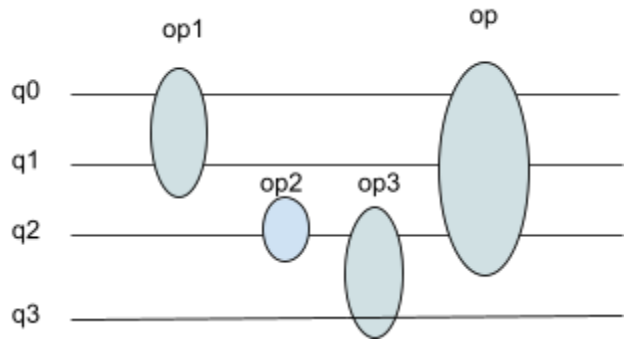
It's better to use pictures to explain Case 2:



Here op* will be merged into op.

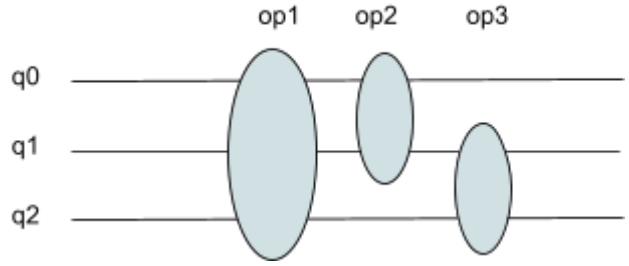


Here L = {op2} and op2 is merged into op. Then L is recomputed as L = {op1} and op1 is also merged into op.

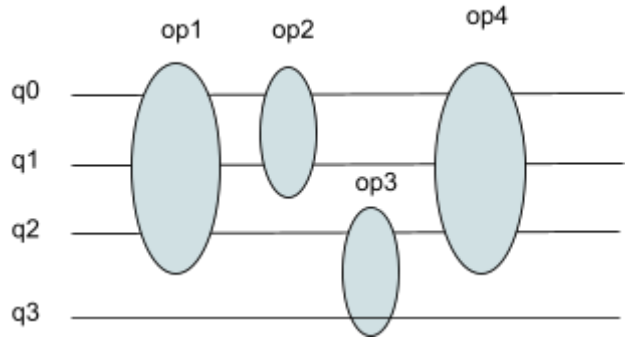


First L= {op3} but op3 can't be merged into op. Then q2 is discarded from future merge attempts for op. Next, L = {op2} and because op2 acts on q2, op2 can't be merged with op. Finally, L = {op1} and op1 can be merged into op.

Let us look now at what happens when both Case 1 and Case 2 are taken into account.



Here op2 is merged into op1 due to Case 1. The result is a bigger CircuitOperation that encapsulates op1 and op2, **and has op1 qubits**. As a result, op3 is now merged into this bigger operation, even though op3 wasn't mergeable with just op2.



Here op2 is merged into op1 due to Case 1, resulting in a bigger operation op* = {op1, op2}. Op3 can't be merged into the left operation op*. Neither can op4. Even though op4 could have absorbed op2 in Case 2, this is now merged into op*, and op* uses qubit q2, blocked by op3.

Algorithm explanation

The result of the algorithm can be described succinctly like this:

- Each merged component has a largest operation op* (in terms of qubit numbers). If there are multiple operations on the same number of qubits, pick the earliest one.
- An operation op is in the component, iff there is a path op = op0 ⊆ op1 ⊆ op2 ⊆ … ⊆ opN = op* so that between opi and op{i+1} there is no other operation, and opi.qubits ⊆ op{i+1}.qubits.
- Merged components are maximal subject to the previous condition.

Then the merged components are uniquely determined by the circuit, and not by the order Case 1 and Case 2 are applied.

Optimization

We don't need to build a CircuitOperation after every merge: we just need to keep track of the qubits of the merged operation, and its moment. This is enough to implement the algorithm.

Extension to is_mergeable

Now imagine there is a method *is_mergeable(op)* which determines if an op is mergeable. For merge_k_qubit_unitaries, *is_mergeable(op)* checks if op is a unitary operation on at most k qubits.

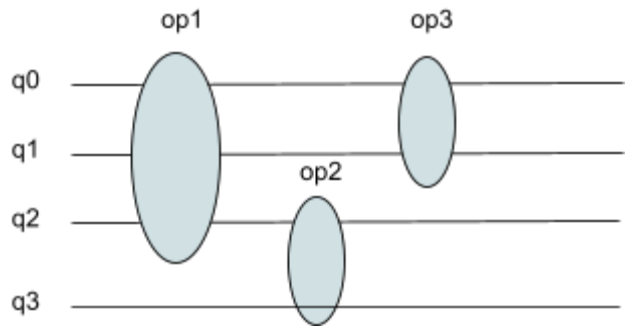
Then this condition can be directly incorporated in the above algorithm: op* that are not mergeable block the shared qubits with op from being considered further. Again, we don't need to create intermediate CircuitOperations.

Extension to merge_func

If an arbitrary merge_func is used, then the resulting components are no longer unique. The result depends on the order merge_func is applied. In this case, building an intermediate CircuitOperation is mandated by the merge_func signature.

The need for bisect.insort in _MergedCircuit

The algorithm uses a data structure _MergedCircuit to represent the merged circuit up to that point.
In Case 1, when op is merged into a previous operation op*, op* is first deleted from _MergedCircuit, op and op* are merged as a new operation op', and then op' is reinserted into op* moment in _MergedCircuit.
The insertion falls back to bisect.insort. This confounded me for a while: why isn't the previous operation op* at the top of the stack?
Then I found an example where this is not the case:



Here op3 will be merged into op1. But qubit_indexes[q2] = {op1.moment, op2.moment}. So when we delete op1, it is not the top of the stack for q2: op2 moment is there now. Also when we insert the merged {op1, op3}, we need to put the moment back in qubit_indexes[q2], which is not at the top.
We need to do deletion and re-insertion: due to merge_func, we can't assume the result will be on the same qubits as op1. Indeed, merge_func(op1, op3) can return op3.