Weighted Partial Matching Algorithm

It is possible to divide our problem into two parts:

- **initial mapping:** finds the initial configuration that our graph will assume;
- **dependency resolution:** from the initial mapping found, we iterate each dependency (a, b) checking if we can use some operation or if we must use swaps.

1 Initial Mapping

The Algorithm 1 presents the pseudo-code of our implementation. In the next paragraph, we will explain it briefly.

In order to find an initial mapping, we first construct a weighted graph $G_d = (V_d, E_d)$ from the dependencies P (line 2). Each vertex $a \in V_d$ represents one logical qubit, and there is an edge $(a, b) \in E_d$ if, and only if, $(a, b) \in P$. Finally, the weight $w(a, b)$ corresponding to an edge $(a, b) \in E_d$ will be equal the number of occurrences of the dependency (a, b) in P.

From G_d, we can define the weight of each vertex a as the sum of the weights of the edges $(a, b) \in E_d$. By calculating these weights, we order the vertices into a sequence V' (line 3). In the lines 5˜21, we run one BFS (Breadth First Search) for each non-allocated vertex. In other words, we start allocating the most used vertex and its adjacent vertices until we finally give up, and assign one physical qubit to each non-allocated logical qubit.

The *findBest* function call in the line 11 finds the best candidate u for the logical qubit a. There are two cases:

- $parent[a] = NULL$ – finds the vertex that best fits from all vertices available;
- $parent[a] = u$ – finds the vertex that best fits from the ones adjacent to u.

2 Dependency Resolution

The dependency resolution algorithm is straight-forward. It starts with the initial mapping f^0 and iterates all dependencies, creating a new mapping f^i for each dependency $i \geq 1$. It is worth noting lines 13 and 22, where we set the first mapping f_0^k and the last mapping f^k.

Algorithm 1 Find a mapping for a sequence of dependencies.

Require: A directed graph $G = (V, E)$ and the sequence of dependencies P.

```
function FindMapping(G, P)
    G_d ← BuildDepGraph(P)
    V' ← order(V_d)
    M ← {}
    for a ∈ V' do
        if visited[a] then
            continue
        end if
        Q ← {a}
        while Q ≠ ∅ do
            a ← pop(Q)
            u ← findBest(G, parent[a], a)
            visited[a] ← true
            M ← M ∪ {a → u}
            for b ∈ G_d.adj[a] do
                if b ∉ Q and ¬visited[b] then
                    push(Q, b)
                    parent[b] ← a
                end if
            end for
        end while
    end for
    return M
end function
```

Algorithm 2 Heuristic algorithm.

Require: A directed graph $G = (V, E)$ and the quantum dependencies P.

```
function SOLVE(G, P)
    Cost ← 0
    f^0 ← FindMapping(G, P)
    for (a_k, b_k) ∈ P do
        (u, v) ← (L(a_k), L(b_k))
        if (u, v) ∈ G.E then
            continue
        end if
        if (v, u) ∈ G.InvE then
            Cost ← Cost + inv
            continue
        end if
        f^k_0 = f^{k-1}
        Swaps ← findSwaps(G, u, v)
        for  do(u_i, v_i) ∈ Swaps
            InsertSwap(k, u_i, v_i)
            a_i ← getAssigned(f^k_{i-1}, u_i)
            b_i ← getAssigned(f^k_{i-1}, v_i)
            f^k_i ← genNewMapping(f^k_{i-1}, a_i, b_i)
        end for
        Cost ← Cost + (len(Swaps) * Swap)
        f^k = f^k_{n_k}
    end for
end function
```