Klaus

Quantum optical setup optimization assisted by propositional logic

---

Setting the problem

Target state

We tested this algorithm to find the graph representation of the GHZ states of any number of parties and local dimension

$$|\,\mathrm{GHZ}(n,\,d)\rangle = \frac{1}{\sqrt{d^n}} \sum_{i=0}^{d-1} \left| i_1 \ldots i_n \right\rangle$$

and highly entangled tri-partite states with different Schmidt Rank Vector (SRV). For states with an odd number of parties, we look for the heralded state, i.e., we tensor-product a |0⟩ state.

```
GHZ[n_, d_] := Table[ConstantArray[i, n], {i, 0, d - 1}]
SRV544 = {{0, 0, 0}, {1, 1, 1}, {2, 2, 2}, {3, 3, 0}, {4, 1, 3}};
SRV644 = {{0, 0, 0}, {1, 1, 1}, {2, 2, 2}, {3, 3, 0}, {4, 1, 3}, {5, 1, 2}};
SRV654 = {{0, 0, 0}, {1, 1, 1}, {2, 2, 2}, {3, 3, 0}, {4, 4, 0}, {5, 1, 3}};
SRV955 = {{0, 0, 0}, {1, 1, 1}, {2, 2, 2},
    {3, 0, 3}, {4, 0, 4}, {5, 0, 5}, {6, 3, 1}, {7, 4, 1}, {8, 4, 1}};
```

Select the target state:

```
state = GHZ[4, 3];
```

The graph will contain as many vertices as number of parties (n) and the possible colors for each edge will be determined by the local dimension (d).

```
If[Mod[Dimensions[state][[2]], 2] ≠ 0, state = Append[0] /@ state];
Print["State basis elements: ", state];
pathmax = Length[state[[1]]];
cmax = Length[DeleteDuplicates[Flatten[state]]];
Print["number of parties (n) = ", pathmax]
Print[ "number of colors (d) = ", cmax]
Print[ "# basis elements = ", Length[state]]
```

```
State basis elements: {{0, 0, 0, 0}, {1, 1, 1, 1}, {2, 2, 2, 2}}
number of parties (n) = 4
number of colors (d) = 3
# basis elements = 3
```

Generate all Perfect Matchings (PM)

- A Perfect Matching (PM) is an independent edge set that contains all vertices of the graph.
- In our problem, the color of each vertex is determined by the color of the incident edge.
- Edges can be bicolored, which implies that a single edge can impose the same color in the connected vertices or two different colors.
- The vertex color combination determines the basis element generated. For example, a color combination of [blue, blue, red, green] generates the state |1102⟩ (if we make the assigment red → |0⟩, blue → |1⟩ and greeen → |2⟩).
- For each color combination, there can be more than one possible PM that generates it. The basis state amplitude will be determined by the sum of all PM that generate that state.

```
t0 = AbsoluteTime[];

(* Name the vertices with letters *)
set = FromLetterNumber[Range[pathmax]];
(* Generate all possible Perfect Matchings (without colors) *)
PMpaths = Flatten[Map[Flatten,
    {Union[Sort /@ (Sort /@ Partition[#, 2] & /@ Permutations[set, {pathmax}])]},
    {-3}], 1];
(* Generate all possible color combinations of the vertices *)
cols = Tuples[Range[cmax] - 1, {pathmax}];
VertexColorings = Flatten[cols[[All, Ordering@#]] & /@ PMpaths, 1];
(* List with all vertex colorings obtained from all PM *)
(* Unique vertex colorings, i.e. deleting the duplicates *)
UniqueVertexColorings = DeleteDuplicates[VertexColorings];

(* Assign a vertex coloring to each PM *)
AllColoredPMsEasyEncoding =
  Flatten[Array[Join[cols[[#2]], PMpaths[[#]]] &, Length /@ {PMpaths, cols}], 1];
(* Divide the PM into pairs (that will correspond to the edge weights) *)
AllWeightsEasyEncoding = DeleteDuplicates[Flatten[
    Map[Flatten[TakeDrop[#, Length[#] / 2] &@Partition[#, 2], {{2}, {1, 3}}] &,
     AllColoredPMsEasyEncoding], 1]];
(* Translate the lists into weights *)
If[pathmax == 4, replace =
   {c1_, c2_, c3_, c4_, a1_, a2_, a3_, a4_} → w[c1, c2, a1, a2] w[c3, c4, a3, a4]];
If[pathmax == 6, replace = {c1_, c2_, c3_, c4_, c5_, c6_, a1_, a2_, a3_, a4_, a5_, a6_} →
    w[c1, c2, a1, a2] w[c3, c4, a3, a4] w[c5, c6, a5, a6]];
If[pathmax == 8, replace = {c1_, c2_, c3_, c4_, c5_, c6_, c7_,
     c8_, a1_, a2_, a3_, a4_, a5_, a6_, a7_, a8_} →
    w[c1, c2, a1, a2] w[c3, c4, a3, a4] w[c5, c6, a5, a6] w[c7, c8, a7, a8]];
If[pathmax > 8, Print["Write the replacement!"]];
AllColoredPMs = AllColoredPMsEasyEncoding //. replace;
AllWeights = AllWeightsEasyEncoding //. {c1_, c2_, a1_, a2_} → w[c1, c2, a1, a2];

t1 = AbsoluteTime[];
TimePMgeneration = t1 - t0;
```

Now we have the list with all possible PM with their corresponding weights and all possible color combinations created by the edges.

The next step consist of identifying those PM that generate the same basis elements and split those basis elements that appear in the target state.

```
t0 = AbsoluteTime[];

AllColoredPMsOrdered = AllColoredPMs[[Ordering@VertexColorings]];
VertexColoringsOrdered = Sort[VertexColorings];
(* Position in the ordered list of those
 PM that generate an element of the target state *)
PosState = Table[Flatten[Position[VertexColoringsOrdered, state[[i]]]],
   {i, 1, Length[state]}];
(* Sum each combination of PM that generate each basis
 element from the target state *)
StateColoringW = Table[Sum[AllColoredPMsOrdered[[PosState[[j, i]]]],
    {i, 1, Length[PosState[[j]]]}], {j, 1, Length[state]}];
(* Repeat the process to identify those PM that generate states
 that do not belong to the target state *)
NoState = Complement[UniqueVertexColorings, state];
PosObstructions = Table[Flatten[Position[VertexColoringsOrdered, NoState[[i]]]],
   {i, 1, Length[NoState]}];
ConstraintsList = Table[AllColoredPMsOrdered[[PosObstructions[[j, i]]]],
   {j, 1, Length[NoState]}, {i, 1, Length[PosObstructions[[j]]]}];
ConstraintsListW = Total[ConstraintsList, {-1}];

t1 = AbsoluteTime[];
TimePMstate = t1 - t0;
```

```
Print["Number of PM: ", Length[PMpaths]];
Print["Number of PM with all color combinations: ", Length[AllColoredPMs]];
Print["Number of different PM (= # of basis elements): ",
  Length[UniqueVertexColorings]];
Print["Total number of weights (edges): ", Length[AllWeights]];
TimePM = TimePMgeneration + TimePMstate;
Print["Δt = ", TimePM / 60, " min"]
```

```
Number of PM: 3

Number of PM with all color combinations: 243

Number of different PM (= # of basis elements): 81

Total number of weights (edges): 54

Δt = 0.00093277 min
```

Logic

Each PM is composed by its edges weights. If one of the weights is zero, then there is no PM. This can be encoded into a logic statement where each weight can take the boolean value of True (if it is different from zero) or False (if it is zero). If only one weight is False, then the full PM is False. The logical clause that encodes each PM is thus

$$w_1 \wedge w_2 \wedge \ldots \wedge w_{n/2}$$

The clauses for this problem are divided into two types:

- State clauses: we need at least one PM that generate each basis element that appear in the target state. Therefore, for each of these elements,

$PM_1 \vee PM_2 \vee ... \vee PM_p = \text{True}$,

$(PM_1 \vee PM_2 \vee ... \vee PM_p)_{\text{basis element}_1} \wedge ... \wedge (PM_1 \vee PM_2 \vee ... \vee PM_p)_{\text{basis element}_L} = \text{True}$

- Obstruction clauses: we can not encode the possible interference between different PM, but we can impose that for each basis element that do not appear in the target state, if all PM except one are False, then the remaining one has to be False as well. This is equivalent to the following clause

$\left(PM_1 \wedge \overline{PM_2} \wedge ... \wedge \overline{PM_p}\right) = \text{False}$

This clause shows that if all PM except one are False (therefore, $\overline{PM} = \text{True}$) then, to obtain False, the remaining PM must be False. We have to ask the same for all combinations of all PM except one being False.
Since we want to solve this problem using a SAT solver (correct outputs evaluate to True), we will ask for the negation of each of these clauses (so the expression will be True when the requirements are fulfilled).
Other possibilities, like two of the PM exist, evaluate the expression to True, since we do not know if it is possible a numerical cancelation between the weights.

```
t1 = AbsoluteTime[];

(* State clauses *)
StateColoring = StateColoringW /. {Times → And} /. {Plus → Or};
(* Obstruction clauses *)
ObstructionFunction =
  Table[Total[MapAt[Not, (Not /@ (# /. {Times → And})), i]] /. {Plus → And},
    {i, 1, Length[PMpaths]}] &;
Obstruction = Flatten[(ObstructionFunction /@ ConstraintsList)];
(* Total clauses *)
Klauses = (Total[StateColoring] /. {Plus → And}) &&
   (Total[Not /@ Obstruction] /. {Plus → And});

t0 = AbsoluteTime[];

Print["Total number of clauses: ",
 Length[StateColoring] + Length[Flatten[Obstruction]]]
Print["Total number of obstructions: ", Length[Flatten[Obstruction]]]
TimeKlauses = (t0 - t1);
Print["Δt = ", TimeKlauses / 60, " min"]
```

```
Total number of clauses: 237

Total number of obstructions: 234

△t = 0.000266170 min
```

Compute the weights that generate the state

To compute what are the weights that generate the state. We will use this function at the very end of Klaus algorithm.

```
AllPMs = Flatten[{ConstraintsListW, StateColoringW}];
Fidelity =
  Abs[Total[StateColoringW]]^2 / (Length[StateColoringW] Total[Abs[AllPMs]^2]);
TotalLoss = (1 - Fidelity);
```

SAT

We can check if the problem with the full connected graph is Satisfible.
Since we have assumed bicolored edges, the result will always be True, since the full-connected graph (all weights = True) is a solution.
The SAT instance with the biggest number of False edges will be the minimal solution. However, finding those instances require on average more time than the heuristic algorithm that we propose in the next section.

```
t0 = AbsoluteTime[];
SatisfiableQ[Klauses, BooleanVariables[Klauses], Method → "SAT"]
t1 = AbsoluteTime[];
TimeFullSAT = t1 - t0;
Print["Δt = ", TimeFullSAT / 60]
```

```
True
```

```
Δt = 0.000133337
```

Klaus algorithm

The full-connected graph is satisfible but we aim to find the minimal graph in terms of the number of edres that can generate the target state. The goal is not to find a solution but rather interpret and understand a solution.
To that aim, we propose an heuristic algorithm, called Klaus.

0. Klaus starts from the full-connected graph. The available edges are all possible edges.
1. Deletes one edge from the available edges at random.
2. Checks if the logic is still satifiable assigning False to those deleted edges and True to the rest of them.
3. If it is, then this edge is “deleted” (set to False). If it’s not, then this edge is removed from the pool of available edges.
4. It repeats the process from 1) until it depletes all available edges.
5. Once it finds a minimal set of edges for which the logic is satisfible, then it proceeds to find the weights of these edges to generate the target state by minimizing the infidelity loss function.

```
t0 = AbsoluteTime[];
```

```
BestKlauses = Klauses;
ReplacementList = {};
BestCurrentWeights = AllWeights;
CurrentWeights = BestCurrentWeights;
AvailableWeights = CurrentWeights;

While[Length[AvailableWeights] > 0,

  (* Select one edge at random from the pool of available edges *)
  falsetmp = RandomInteger[{1, Length[AvailableWeights]}];
  deleteedge = AvailableWeights[[falsetmp]];
  (* This edge will not be available for the next iteration,
  either because it can be set to False or because it has
   to exist or the expression will not be satisfiable *)
  AvailableWeights = DeleteCases[AvailableWeights, deleteedge];

  (* We impose that edge as False and compute
   the total logical expression from the previous iteration *)
  CurrentKlauses = BestKlauses /. Thread[deleteedge → False];

  (* Check if the expression is still satisfiable *)
  logicVal =
   SatisfiableQ[CurrentKlauses, BooleanVariables[CurrentKlauses], Method → "SAT"];

  If[logicVal == True,
   (* Remove it
    (append it to the ReplacementList where previous removed edges are *)
   AppendTo[ReplacementList, deleteedge];
   CurrentWeights = Complement[AllWeights, ReplacementList];
   BestReplacementList = ReplacementList;
   BestCurrentWeights = CurrentWeights;
   BestKlauses = CurrentKlauses;

   (* if logicVal = False, that weight is necessary to SAT=True! keep it,
   i.e. it's not available for removing it! *)
  ];

 ];
t1 = AbsoluteTime[];
TimeLogicOpt = t1 - t0;

Print["Logic reduction completed."];
Print["Solution = ", Length[BestCurrentWeights], "/",
 Length[AllWeights], " weights, Δt = ", TimeLogicOpt / 60, " min"]

t0 = AbsoluteTime[];

(* Once we have the logical minimal solution,
we minimize the infidelity of the state generated by the
 remaining PM w.r.t. the target state to obtain the weights *)
(* Deleted edges will be set with 0 weights *)
CurrentLoss = TotalLoss /. Thread[BestReplacementList → 0];
initvar =
  Transpose[{BestCurrentWeights, RandomReal[{-1, 1}, Length[BestCurrentWeights]]}];
sol = FindMinimum[CurrentLoss, initvar, AccuracyGoal → 3, PrecisionGoal → 3];
```

```
t1 = AbsoluteTime[];
TimeOpt = t1 - t0;

fid = Fidelity //. sol[[2]] //. Thread[BestReplacementList → 0];
Print["Optimization completed. "];
Print[ "Fidelity = ", fid, ", Δt = ", TimeOpt / 60, " min"];

If[fid < 0.9, Print["Logic failed. A crucial edge was deleted. Try again."],
  Print["Target state found!"]];

(* Print the solution *)
coef = (AllColoredPMs //. sol[[2]] //. Thread[ReplacementList → 0]);
nonzero =
  Flatten[SparseArray[AllColoredPMs //. sol[[2]] //. Thread[ReplacementList → 0]][
    "NonzeroPositions"]];
norm = Sum[coef[[nonzero[[i]]]]^2, {i, 1, Length[nonzero]}];
resstate = 0;
Do[coeftmp = Chop[coef[[nonzero[[i]]]] / Sqrt[norm], 10^(-3)];
  If[coeftmp ≠ 0, resstate = resstate + coeftmp (AllColoredPMs[[nonzero[[i]]]]) //.
     w[cc1_, cc2_, a_, b_] → a[cc1] * b[cc2]], {i, 1, Length[nonzero]}];
Print["Target state ="];
Print[state];
Print["Obtained state = "];
Print[Chop[resstate]];
Print["Graph weights:"]
Print[Chop[sol[[2]]]]
Print["Total time optimization = ", (TimeLogicOpt + TimeOpt) / 60, " min"]
Print["Total time(problem generation + optimization) = ",
 (TimePM + TimeKlauses + TimeLogicOpt + TimeOpt) / 60, " min"]
```

```
Logic reduction completed.

Solution = 6/54 weights, △t = 0.001378875 min

Optimization completed.

Fidelity = 0.999999, △t = 0.000400010 min

Target state found!

Target state =

{{0, 0, 0, 0}, {1, 1, 1, 1}, {2, 2, 2, 2}}

Obtained state =

0.576703 a[0] b[0] c[0] d[0] + 0.577314 a[1] b[1] c[1] d[1] + 0.578034 a[2] b[2] c[2] d[2]

Graph weights:

{w[0, 0, a, d] → 0.655925, w[0, 0, b, c] → 0.831031, w[1, 1, a, c] → 0.964865,
 w[1, 1, b, d] → 0.565542, w[2, 2, a, b] → -0.707523, w[2, 2, c, d] → -0.772203}

Total time optimization = 0.00177889 min

Total time(problem generation + optimization) = 0.00297783 min
```