Notes on: the blockBP tensor-network contraction algorithm

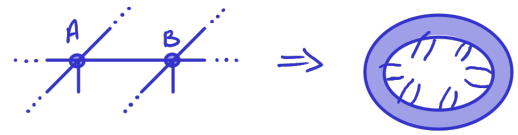
Itai Arad

December 12, 2021

1 Introduction — General idea

The blockBP algorithm for tensor-network (TN) contraction is a generalization of the BP algorithm for TN contraction [AA21], applied to blocks of vertices in the TN instead of single vertices.

Our goal is to contract a 2D PEPS tensor network. Specifically, we want to be able to calculate the local environment of local tensors or nearest neighboring tensors:

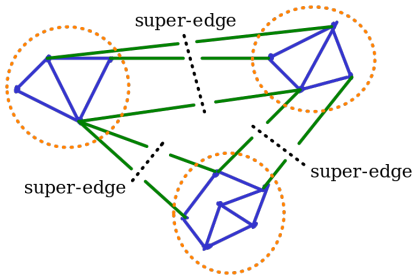


original A, B tensors local env' of A, B

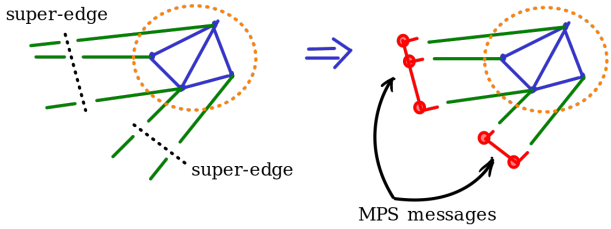
We first pass to the $\langle\psi|\psi\rangle$ TN by contracting $|\psi\rangle$ with $\langle\psi|$ along the physical legs. The resulting TN is then legless and can be described by a graph $G = (V, E)$, where for every $v \in V$ there is a tensor T_v which is the contraction of the original bra-ket tensors along the physical leg. Similarly, every edge correspond to a double edge of the original PEPS. We then partition the system into blocks, where blocks are non-overlapping contiguous subsets of particles. The larger the blocks are, the better the approximation becomes, at the cost of increased memory and CPU resources.

The partition of the system into blocks defines a coarse-grained graph $\underline{G} = (\underline{V}, \underline{E})$, where $\underline{V}$ is the set of blocks and $\underline{E}$ are the *super-edges* connecting the blocks. Each super-edge is an ordered set of edges connecting two blocks[1]:

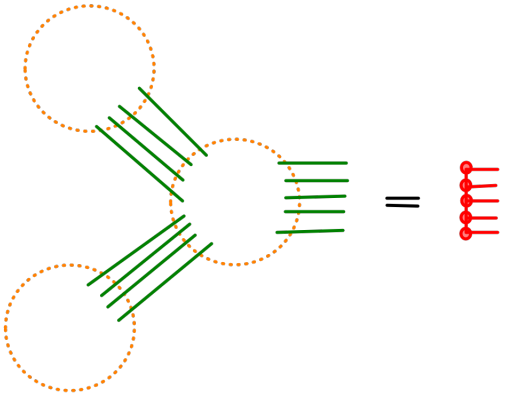
[1] The ordering is a result of our assumption that the the TN is a 2D TN.



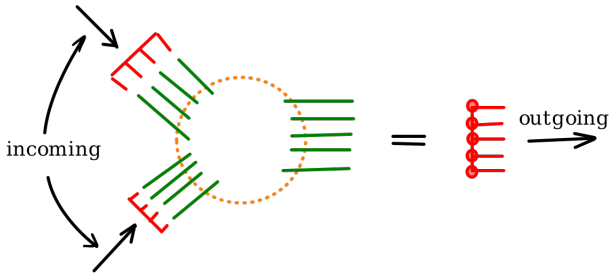
The algorithm is essentially the BP algorithm done on the coarse-grained TN, where the blocks replace the local tensors and the super edges replace the edges. However, since the Hilbert size of the coarse-grained objects is very large, we replace the contraction of the vertex tensors by approximate contraction and the messages along the super edges are replaced by MPS messages. Each MPS message is an MPS whose legs correspond to the edges that make up the super edge. Along each super-edge there are two messages going in opposite directions. The algorithm ends when the messages converge to some fixed point. Once at the fixed point, the MPS messages, together with the block define a small TN, from which the environment can be calculated by direct contraction:



Just as in the ordinary BP algorithm, when the graph $\underline{G} = (\underline{V}, \underline{E})$ is a tree, the algorithm becomes exact (up to the inaccuracies in the contraction of the messages) and the messages have a simple interpretation. Each super-edge bi-partites the system into two, and the MPS messages along it are the contraction of the two parts of the TN.



Also, just as in the ordinary BP, an outgoing message from a block to another block is created by contracting all other incoming messages to that block with the tensors of the block itself:



1.1 A broad overview of the algorithm

Given a $\langle\psi|\psi\rangle$ TN, together with a partition to blocks and super-edges, together an initial set of MPS messages along them, the algorithm is a essentially the following loop:

alg:s1

1. Go over the blocks in some order
2. For each block, take its incoming MPS messages, and produce the outgoing messages. This is done approximately using the boundary-MPS method.
3. The outgoing messages of that block then become the incoming messages of the neighboring blocks.
4. Once all blocks have been visited, calculate the distance between the current MPS messages and the messages of the previous round.
5. If the distance is smaller than some δ, exit. Otherwise, go back to step 1

2 Detailed description of the algorithm

Here we provide a more detailed description of the `blockbp.py` module and the `blockbp` function in it.

2.1 Input Parameters

The `blockbp` function is defined by:

```
def blockbp(T_list, edges_list, pos_list, blocks_v_list, sedges_dict,
sedges_list, blocks_con_list, PP_blob=None, m_list=None,
classical = False, D_trunc=None, D_trunc2=None, eps=None, max_iter=100,
    delta=1e-8,Lx=1e9, Ly=1e9):
```

Below, we describe its input parameters.

1. `T_list` — The list of tensors that make up the entire TN. Just as the input for the `ncon` function, its a list of `numpy` arrays.

2. `edges_list` — A list describing the edges connected to every tensors, using the `ncon` convention. This is a list of lists. For every tensor in the TN, there is a list of edges connected to it, ordered in accordance to the indices in the array that represents it. Each edge is a label; it can be anything that can be in a list — an integer, a string etc. Tensors that are contracted together along some edge will have that same edge label appears in their lists.

   Note: Unlike the `ncon` case, here we do not expect any external "open" legs. So every edge label must appear in exactly two tensors.

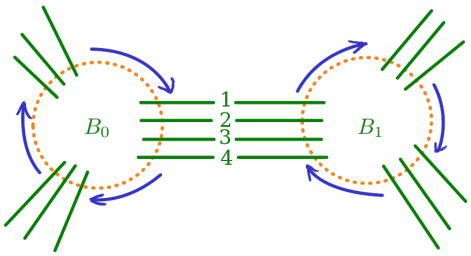
3. `pos_list` — This is a (x, y) position of every tensor. Every entry is a tuple of two numbers. The (x, y) do not have to describe exactly where each tensor sits — this is of course an irrelevant information. The positions are used to determine the angles of the edges with respect to the vertices, which is needed for the `bubblecon` function. Therefore, any placement of the tensors on the plane will be fine, as long as it preserves the 2D topology of the graph, i.e., as long as edges as straight lines do not cross each other.

4. `blocks_v_list` — This is the list of blocks. It is a list of lists. The k'th list is the list of the tensors that make up the kth block. Every item on that list is an ID of the tensor in that block, as defined by `T_list`.

5. `sedges_dict` — This is a dictonary that defines the super-edges. Every key in the dictonary is a label of a super-edge, and the value of each key is a list of edges that make up the superedge. The edges in the list must appear in the o rder that they appear in the graph. For example, if we edges $1, 5, 8$ make one super-edge and edges $2, 4, 6$ make another, then `sedges_dict` may contain the entries `{ 'A':[1,5,8], 'B':[2,4,6]}`, where "A" is the label we assign with the first super-edge and "B" with the second.

   Note: This parameter is essentially redundant, as it can be deduced from the other parameters. This might be done in future releases.

6. `sedges_list` — This is the list of super edges that are connected to every block. The list has an item for each block, corresponding to the entries in `blocks_v_list`. Each item on the list is a list of tuples of the form `[(se1,or1), (se2,or2), …]`. Evey tuple is made of two parts: the first is the label of the superedge. The second is a ± 1 integer that describes the ordering of the edges in the super-edge with respect to the block. It can be either clockwise with respect to the block $(+1)$ or counter-clockwise (-1). Consider, for example, two blocks B_0, B_1, connected by the super-edge '`se1`' made of the edges $1, 2, 3, 4$, which is defined in `sedges_dict` by `{…, 'se1':[1,2,3,4], …}`.



Then in the `sedges_list`, at the B_0 item there will be a tuple `('se1', 1)` because the edges $1, 2, 3, 4$ are ordered in a clockwise fashion with respect to B_0, while at the B_1 item there will appear `('se1', -1)`.

Note: Like the previous parameter, also this parameter is essentially redundant, and can be deduced from the other parameters.

7. `blocks_con_list` — This is a list of the contraction order inside each block that is needed to give the `bubblecon` algorithm in order to calculate each one of the outgoing MPS messages. It is a list of lists. The kth item on the list is the list of contraction orders of the kth block. If the kth block is connected to ℓ adjacent super-edges, then it will contain ℓ contraction orders — each for computing the outgoing MPS message of every adjacent super-edge. Finally, every contraction order contains the order of contracting the TN that creates the outgoing MPS. That TN is made of two types of vertices:

   - The original vertices of the block.
   - Vertices of the incoming MPS messages on all other external edges that are not part of the outgoing super-edge.

   These vertices are denoted by tuples of the form `(type, id)`. When the vertex corresponds to an original block vertex, the `type` variable is 'v', and the `id` variable is the id of that vertex in the global TN. When it comes from an incoing MPS then `type` variable is 'e', and the `id` variable denotes the edge to which the MPS vertex is connected.

8. `PP_blob=None` — This is a dictionary that can optionally hold some pre-processing information. The dictionary is also an output parameter of the function. So if we run blockbp as an iterative process, in which only the tensors in `T_list` change (but not the connectivity of the network or the dimensions), we can supply it as input and save much of the pre-processing steps.

   `PP_blob` contains the following entries: `sedges_list`, `blocks`, `mps_mapping`, `mpsD`, `BTN_list` All of which are calculated in the pre-processing step.

   Note that non of the above objects depends on the tensors in `T_list`, except for `BTN_list`. So if `blockbp` is invoked with `PP_blob`, then the tensors in `BTN_list` are updated from the tensors in `T_list`.

9. `m_list=None` — This is the initial list of messages. It is a double list of MPS objects in which `m_list[i][j]` contains the MPS message between block i to block j. If it is omitted then the algorithm starts with random MPS messages (that are also PSD).

10. `classical=False` — This flag tells `blockbp` if the underlying TN is a quantum double-edge model or is a single-edge classical model. This only matters if no initial MPS messages are given. In such case `blockbp` has to create initial MPS messages, and here the two cases differ: quantum double edge initial messages are MPS that represent product states of a vectorization of the identity — this corrresponds to ket-bra contraction of the double edge. On the other hand, classical initial messages correspond to product states of the uniform probability distribution.

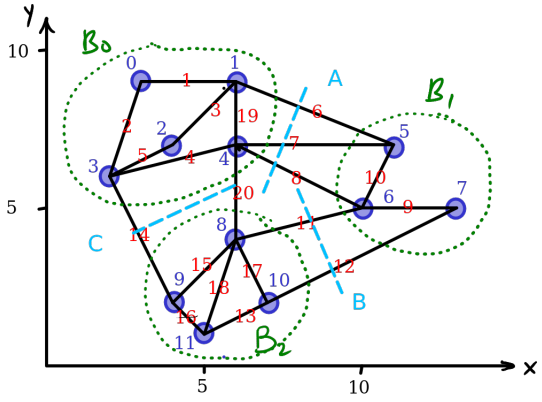
11. `D_trunc=None, D_trunc2=None, eps=None` — These are the boundary-MPS truncation parameters that are passed directly to the `bubblecon` contraction routine.

12. `max_iter=100, delta=1e-8` — Stopping criteria for the BP iteration. Either when the number of iterations reaches `max_iter` or when the average distance between sets of consecutive messages is $<$ `delta`.

13. `Lx=1e9, Ly=1e9` — The periods of the 2D lattice in the X direction and in the Y direction. The periodicity only comes to play when the X,Y distances between two neighboring blocks exceed `Lx/2` or `Ly/2`.

2.2 Example of input parameters for a simple model

Here we give an example of a the input parameters for a simple model made of 12 spins, which are partitioned into 3 blocks, B_0, B_1, B_2 with 3 super-edges between them labeled as A,B,C.



In the figure above, the IDs of the tensors are integers 0,1,2,...,11 written in blue. The edges are labeled also by integers, and are typed in red. The super-edges are denoted by lightblue.

The non-trivial parameters in this case are:

```
#              T0     T1      T2    T3     T4      T5     T6      T7
pos_list = [ (3,9), (6,9), (4,7), (2,6), (6,7), (11,7), (10,5), (13,5),
#   T8     T9     T10    T11
  (6,4), (4,2), (7,2), (5,1)]

#               T0        T1       T2       T3          T4
edges_list = [ [1,2], [1,3,19,6], [5,3], [2,5,4,14], [19,4,7,8,20],
#   T5           T6        T7          T8               9
[6,7,10], [8,10,9,11], [9,12], [11,20,15,18,17], [16,14,15],
#  10            11
[13,17,12], [16,18,13]]

#                     B0          B1        B2
blocks_v_list = [ [0,1,2,3,4], [5,6,7], [8,9,10,11] ]

sedges_dict = { 'A':[6,7,8], 'B':[11,12], 'C':[14,20]}

#                         B0                      B1                 B2
sedges_list = [ [('A', 1), ('C', -1)], [('A',-1), ('B',-1)], [('B',1), ('C',1)]]
```

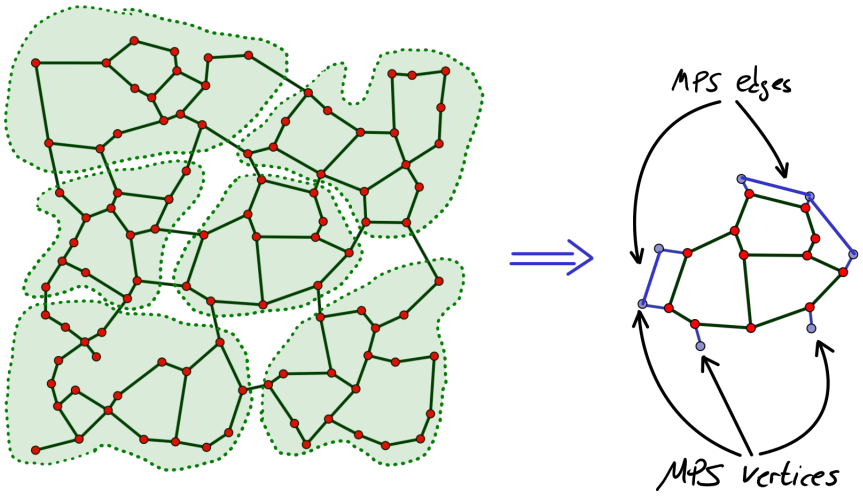
The contraction order in the different blocks can be

```
bcon0 = [ ([('e',14),('e',20),('v',3),('v',4),('v',2),('v',0),('v',1)],\
  pi/2), \
  ( [('e',6),('e',7),('e',8),('v',1),('v',4),('v',0),('v',2),('v',3)],pi)]

bcon1=[ ([('e',11),('e',12),('v',7),('v',6),('v',5)],3*pi/2), \
      ([('e',6),('e',7),('e',8),('v',5),('v',6),('v',7)],0)]

bcon2=[ ([('e',14),('e',20),('v',9),('v',11),('v',8),('v',10)],3*pi/2), \
        ([('e',11),('e',12),('v',10),('v',8),('v',11),('v',9)],pi)]

blocks_con_list = [bcon0, bcon1, bcon2]
```

2.3 General structure of the algorithm

2.3.1 Block Tensor Networks (BTNs)

To perform the Block BP, the algorithm uses *Block Tensor Networks* (BTNs). A block tensor network is the TN of the block which is derived from the global TN. We add to it the tensors of *all* the incoming MPS messages that are contracted to its external edges:



Together, they form a *closed* TN that is characterized by the following variables:

- `T1_list` — The list of tensors in the BTN (including the tensors of the MPS messages)
- `edges1_list` — BTN connectivity. For each tensor in `T1_list` there is a list of edges connected to it.
- `angles1_list` — The list of angles of the edges of every tensor.
- Contraction orders `bcon` — If a block has k adjacent super edges, then we need k contraction order, one for each super-edge. The contraction order for the superedge `se` is the contraction order for the `bubblecon` for the open TN we get by removing all the MPS vertices of the `se` superedge. This is TN defines the outgoing message from our block to the block that is adjacent to it via `se`.

  The definition of the contraction orders are given initially by the input parameter `blocks_con_list`, but we need to rewrite them in terms of the internal vertices of the BTN.

  For each BTN, `bcon` is therefore a list of contraction orders.

Note that all these variables are compatible with the TN specification that is fed into the `bubblecon` function (and therefore also with `ncon`).

The information of every BTN is then stored by a tuple

```
BTN = (T1_list, edges1_list, angles1_list, bcon)
```

All these tuples are stored in the list `BTN_list`.

In addition to the BTN objects, we need two other lists for the main BP loop:

1. The `mps_mapping[ib][jb]` doubel list. This is a mapping that maps the tensor of every incoming MPS message to its place in vertex in the BTN. The `ib,jb` element of the double list holds the mapping of the MPS message `ib`→`jb` from the `ib` block to the `jb` block. The mapping is a list with the length of the MPS, where every item is the internal number of the tensor in the BTN to which it is mapped.

2. The `blocks[se]` dictionary. This dictionary gives the `(ib,jb)` blocks ID that define every super-edge `se`.

2.3.2 The Pre-Processing step

In the pre-processing step, we take the input parameters and derive from them the `BTN_list` and the `mps_mapping` double list. This is done using the following steps:

1. Use the `edges_list` to create the `vertices` dictonary in which the edges are the keys and for every edge the value is the tuple (i, j) of its adjacent tensors.

2. Do the same thing for the super edges: use the `sedges_list` to create the `blocks` dictonary.

3. Calculate the angles of every tensor from the (x, y) positions.

4. Initialize the `mps_mapping[ib][jb]` double list as an empty skeleton.

5. Main loop for creating the `BTN_list`. Go over all blocks:

   (a) Let `v_list` be the list of tensors in the block (take it from `blocks_v_list`).

   (b) Use `v_list` to create `T1_list, edges1_list, angles1_list` by restricting the global lists to the block vertices.

   (c) initialize the dictionary `vertices_dict`, which will be used to map the vertices in the TN to their corresponding veritces in the BTN.

   It also maps the external edges of the BTN to their corresponding MPS vertices in the BTN. The key format of `vertices_dict` corresponds to the way that the block contraction-order is given in the input parameter `blocks_con_list`, i.e., either `('v',v)`, or `('e',e)`.

   (d) Add to `vertices_dict` all the TN vertices that participate in the BTN. The key is of the form `('v',v)` where `v` is the ID of the vertex in the TN, and the value is the ID of the vertex in the BTN.

(e) Go over all super-edges of the BTN, and for each superedge go over all of its edges, depending if its in clockwise orientation or anti clockwise:

i. Add the middle of every such edge as an empty place holder to `T1_list`. This defines an ID for the MPS vertex in the BTN.

ii. Add this ID to the appropriate place in the `mps_mapping` list.

iii. Add the external edge as a key `('e', e)` to the `vertices_dict`, where the value is the ID of the MPS vertex.

iv. Calculate the position of the new vertex as the exact middle of the edge.

v. Finally, also add the edges of the MPS messages to `edges1_list` (and update their angles in `angles1_list`).

(f) Finally, once we've gone over all the super-edges of the block, we use the `vertices_dict` to translate the input parameter `blocks_con_list` to the `bcon` list that specifies the different contraction orders of the BTN.

At the end of the Pre-Processing step, we are left with:

- `BTN_list`
- `mps_mapping`
- `blocks`

All of these are used in the main BP loop.

2.4 Periodic Boundary conditions

`blockbp` is capable of running in periodic boundary conditions (BC). The idea is simple. Let L_x, L_y are the periods of the lattice in the $\hat{x}, \hat{y}$ directions. Then we demand that every two vertices in a block are sufficiently close to each other in the sense that

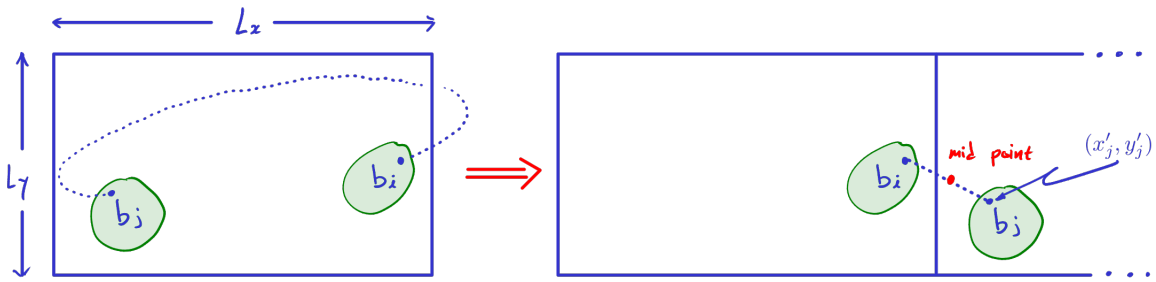
$$|\Delta x| = |x_1 - x_2| < L_x/2 \quad \text{and} \quad |\Delta y| = |y_1 - y_2| < L_y/2.$$

So when we contract a block to find the outgoing messages, the vertices within the block can be assumed to be in the usual open boundary condition. The only possible complication comes from the incoming MPS vertices. These are located in the middle of the edges that connect two blocks. Assume then that block b_i and b_j are connected by a super-edge, and assume that we are calculating the outgoing messages of block b_i. Let $(x_i, y_i) \in b_i$ and $(x_j, y_j) \in b_j$. We wish to calculate the location of the middle point connecting (x_i, y_i) with (x_j, y_j). Then we redefine the coordinates of

(x_j, y_j) as follows:

$$x'_j = \begin{cases} x_j + L_x, & \text{if } x_i - x_j > L_x/2 \\ x_j - L_x, & \text{if } x_i - x_j < -L_x/2 \\ x_j, & \text{otherwise} \end{cases}$$

In a similar way we define y'_j. Once we have (x'_j, y'_j), we can define the middle point as $(\frac{x_i+x'_j}{2}, \frac{y_i+y'_j}{2})$.



References

[AA21] R. Alkabetz and I. Arad. Tensor networks contraction and the belief propagation algorithm. *Phys. Rev. Research*, 3:023073, Apr 2021.