Nested block-sparse random testing based on BFSRANDOM method

Junjie Pan

The concept of block-sparse originated from signal processing research area[1], which facilitate many real-world applications especially for dimension deduction problem. One simple case of block-sparsity is structure-less block sparsity. That is, for example, given an array arr(which can be a list, a vector, or a tuple) of n length with sparse-level k (which means only n*k elements of arr store the useful information we want) where 0<k<= 0.5 in general, when we divide the array into b blocks, then the n*k useful information would also sparsely diffuse in u blocks of totally b blocks, where u/b<= 0.5.

Take bug testing as an example, suppose there is a tuple consist of both safely actions and unsafely actions, and the unsafely actions are just small part of all actions, then the unsafely actions(we use bug to denote unsafe action in the remain contents) can be viewed as very sparse information of the tuple. Then, based on the block-sparsity theory, when we divide tuple into several blocks with the block size satisfying some criteria (actually this is a bit of complicate, so I temporally don't consider it and simply set it as fix number in this work), then the bugs would diffused sparsely in this blocks, which means the number of blocks that would contain the bugs would sparse compared to the total number of blocks, with high probability.

Actually the hardest part of block sparsity is its mathematical proof part, however, the algorithm part is instead simple. Let's see my pseudo-code for its application on the testing case.

Consider the sparse properties would be not so good when the length of queue is small, I adjust the sparsity level based on different length.

Algorithm:

```
Define n as length of queue # queue is as the same definition of bfsrandom
Define block_num # number of blocks
Define sparse_level
Define valid_blks =    n*sparse_level # the part of blocks that would contain bugs
Define ref_tb # reference table with n length, ref_tb[i] = 1 indicate there would be bug
                # in the i-th actions and test the i-th bundle of actions

If   n >= 2*block_num
      Block_size = int ()

      Valid_ids = indexes of randomly chosen valid_blks from block_num of blocks

      For   i   in valid_ids
            start_id = i*block
```

```
            end_id = i*(block+1)
            ref_tb[start: end] = inner_rand(end_id- star_id, in_spar_level, min)

else:
    ref_tb = inner_rand(n, 0.5, 1)


 inner_rand(len, s_level, min_num):

    arr = [0]*len
    sparsity = int(len*s_level);
    if sparsity< min_num:
        sparsity = min_num;

    a = range(len);
    random.shuffle(a);
    sparse_ids = a[0:sparsity]#[random.randint(0, len-1) for i in range(sparsity)]
    print sparse_ids
    for i in sparse_ids:
        arr[i] = 1;

    return arr
```

Actually, I call this algorithm nested-block, because beside divide the tuple as blocks and test only part of blocks based on sparsely random chosen ones, I also, nearly, sparsely choose elements of the valid blocks, just as the function inner_rand defined in the above. However, when the length of current queue < 2*blocks_num, I just use inner_rand function to get the number of elements to be tested.

Actually, the motivation of this algorithm based on the assumption that bugs are only very very small part of whole actions. Therefore, bugs are very sparsely diffused. Therefore, I think test all the action in order maybe wast too much time to test the actually right codes part, and we can actually make use of the time spend on test right code part on the part that likely have bugs. Therefore, in terms of detecting sparse bugs from a large action pools, it become kind of signal processing problem.

Conclusion:

Actually, there are many things need to be improved, such as block the actions from the actions level instead of from the queue level, which is much harder. In the future, I will try to erase the timing methods for testing, instead, I will try to make use of block-sparsity properties to determined when to stop testing.